This is to certify that the

dissertation entitled

Computer Aided Process Planning:

Task Representation and Sequencing

presented by

Joseph M. Miller

has been accepted towards fulfillment
of the requirements for

Doctoral degree in Computer Science

J C Stockman
Major professor

Date 7 Sep 1990

Computer Aided Process Planning: Task Representation and Sequencing

By

Joseph Michael Miller

A DISSERTATION

Submitted to
Michigan State University
in Partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science

1990

Abstract

Computer Aided Process Planning:
Task Representation and Sequencing

By

Joseph Michael Miller

Process planning is a combination of art and engineering concerned with organizing manufacturing operations to turn raw materials into a product. The computer offers the process planning task increased potential for standardization, optimization, and more rapid response to changes in part design or in the manufacturing conditions on the factory floor. This thesis investigates the representation and sequencing of tasks in process planning. Unfortunately, a large number of possible process plans can make finding efficient plans very difficult.

An Automated Assembly Planning with Fasteners (AAPF) system is described that uses geometric reasoning to derive feasible assembly sequences based on the geometric data present in a CAD model. AAPF is the first automated assembly planning system to pay attention to individual fasteners. It automatically derives which components in a product are held together by which fasteners, and uses this connectivity information and the type, size, orientation, and location of fasteners to aid in determining efficient assembly plans.

An algorithm is presented that acts as an oracle in determining the number of possible task orderings using a precedence graph representation for a set of tasks and precedence constraints. The Oracle uses three subgraph reduction patterns to find the exact number of possible task orderings for a class of *N-fold* graphs or an upper and lower bound on the number of orders for all other precedence graphs. An analysis of the Oracle algorithm shows that an exact answer or bounds on the number of possible orders is calculated in $O(V^2 * E)$ time and $O(V^2)$ space.

A CAPP system for Concurrent Design and Evaluation (CDE) is proposed but not implemented. CDE takes advantage of the Oracle algorithm and two interactive design critics to help find efficient process plans. Although some components of the CDE system exist, others require additional research into difficult problem solving areas such as: path and grip planning, determining stability of objects, and incorporating inspection and testing into plans.

Publication Information

Earlier versions of parts of this dissertation have appeared in the following publications:

1) *"Precedence Constraints and Tasks: How Many Task Orderings ?"*, Joseph Miller, George Stockman, 1990 IEEE International Conference on Systems Engineering, August 9-11, 1990, Pittsburgh, Pennsylvania, pages 408-411.

2) *"The Number of Linear Extensions in a Precedence Graph"*, Joseph Miller, George Stockman, 1990 IEEE International Conference on Robotics and Automation, May 13-18, 1990, Hyatt Regency, Cincinnati, Ohio, Vol. 3, pages 2136-2141.

3) *"Determining Search Space Size in Precedence Graphs"*, Joseph M. Miller, George C. Stockman, First Great Lakes Conference, October 18-20, 1989, Western Michigan University, Kalamazoo, Michigan.

4) *"Computer-Aided Process Planning: An Introduction and Overview"*, Joseph Miller, George Stockman, PRIP Lab Technical Report No. PRIP-89-3, February 1989, pp. 62.

5) *"Automated Assembly Planning with Fasteners"*, Joseph Miller, Richard Hoffman, 1989 IEEE International Conference on Robotics and Automation, May 14-19, 1989, Registry Resort Scottsdale, Arizona, Vol. 1, pages 69-74.

6) *"WoodMaster: A Process Planner for Wooden Objects Made up of Block and Cylindrical Components"*, Joseph Miller, Case Center Technical Report, April 1988, pp. 65.

To my parents,

and my wife, Bev

ACKNOWLEDGEMENTS

I wish to thank my thesis advisor Professor George Stockman for his time, effort, guidance, and the occasional canoe outing. Without his direction and encouragement this endeavor would not have been possible. Many thanks are due to my committee members Professors Erik Goodman, William McCarthy, and Anthony Wojcik for their guidance and support. Special thanks to Dr. Goodman for directing me during an independent study course on Computer Aided Process Planning and for making the facilities of the Case Center available to me. I also wish to thank Dr. Wojcik for supervising me during an independent study course on Artificial Intelligent Languages. Over the course of my studies at Michigan State I received help and direction from Drs. Dubes, Esfahanian, Geske, Hsu, Ni, Jain, Sticklen, and Tuceryan. Additionally, I gratefully acknowledge all the other faculty members who have made my educational experience both worthwhile and interesting.

Many kudos to H. R. Keshavan and Richard Hoffman at the Northrop Research and Technology Center for their support and help -- especially, during my summer internship in 1988. I thoroughly enjoyed and academically benefited from my Northrop experience. I appreciated the thought provoking discussions, snail mail, and e-mail correspondence that I had with Thomas DeFazio, Rick Hoffman, Luiz Homem de Mello, Peter Winkler, Jan Wolter, and Yifei Huang. This thesis was greatly enhanced by their feedback and helpful hints. Of course, the responsibility for the contents of this dissertation remains with me.

This research has been supported in part by the National Science Foundation grant #CDA-8806599, the Northrop Research and Technology Center, and Michigan State University. Special thanks to the Department of Computer Science for supporting me with a Graduate Assistantship throughout my educational experience and for the occasional travel funds that allowed me to present my research results and attend various conferences. I appreciate Dr. Anil Jain's support and encouragement during my years as a PRIP lab System Manager. Over my academic career I was fortunate to qualify for a three year Distinguished Dean's Fellowship and several G.T.E. Fellowships which relieved me from the worry of how to pay my bills and allowed me to concentrate more fully on my research.

I must acknowledge the support and kinship of all former and current graduate students at Michigan State, especially Rick Hoffman, Gongzhu Hu, Bruce McMillin, Sei-Wang Chen, Patrick Flynn, Yhwyng Lee Hoffman, Farshid Farrokhnia, Deborah Trytten, Narayan Raja, Greg Lee, Arun Nanda, Tim Newman, Eugene Wallingford, Mahmoud Pegah, John Kern, and Satish Nadabar. The weekly PRIP and AI/KBS meetings gave me an opportunity to learn more about the fascinating wealth of information in these areas. Additionally, the occasional Chocolate Cake party or trip to the M.S.U. Dairy Store for ice cream helped to make graduate life more bearable. Many special thanks to Dave Marks for carrying out and improving the system management of the PRIP lab facilities.

Last but foremost, I acknowledge the love and support of my parents, family and especially my wife, Bev. Additionally, I thank my father, Dr. Raymond Miller (the original "Dr. Miller"), for being such a positive role model. Finally, I acknowledge the many opportunities and blessings I have received directly and indirectly from my relationship with the Lord who truly made all of this possible.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Process planning is a combination of art and engineering concerned with organizing manufacturing operations to turn raw materials into a product. Process planning requires the application of manufacturing knowledge, including knowledge about machines, tools, geometry, physics, manufacturing procedures, and costs associated with manufacturing the product. The computer offers increased potential for standardization, optimization, and more rapid response to changes in part design or in the manufacturing conditions on the factory floor. Products should be designed with manufacturability in mind, and ideally, feedback from the process planning task would be incorporated into the final design.

An overview of computer-aided process planning (CAPP) is given in this chapter. Process planning can be performed in many different domains such as composite parts, plastics, sheet metal, or discrete metal parts. Automatic process planning systems are developed for specific domains and for a given factory environment. After defining process planning, the role of CAPP in Computer Integrated Engineering is discussed. Next the manufacturing cycle is outlined to show how process planning is of critical importance in creating a profitable product. Section 1.3 reviews the two main approaches used in computer-aided process planning. *Variant* process planning is currently the most common method since it builds upon existing databases of parts that are organized into classes based on group technology (GT) codes. The *generative* approach encodes the knowledge of a process planner into a set of rules that are used to automatically generate a process plan from a part description.

1.1. What is Process Planning

Process planning involves taking product specifications, usually an engineering drawing (or as of late, the information in a CAD model), and producing a set of manufacturing plans. Process plans are usually created by a process planner who is typically a production engineer or an expert machinist with many years of experience. The final process plan contains the steps necessary to transform raw materials into the desired product.

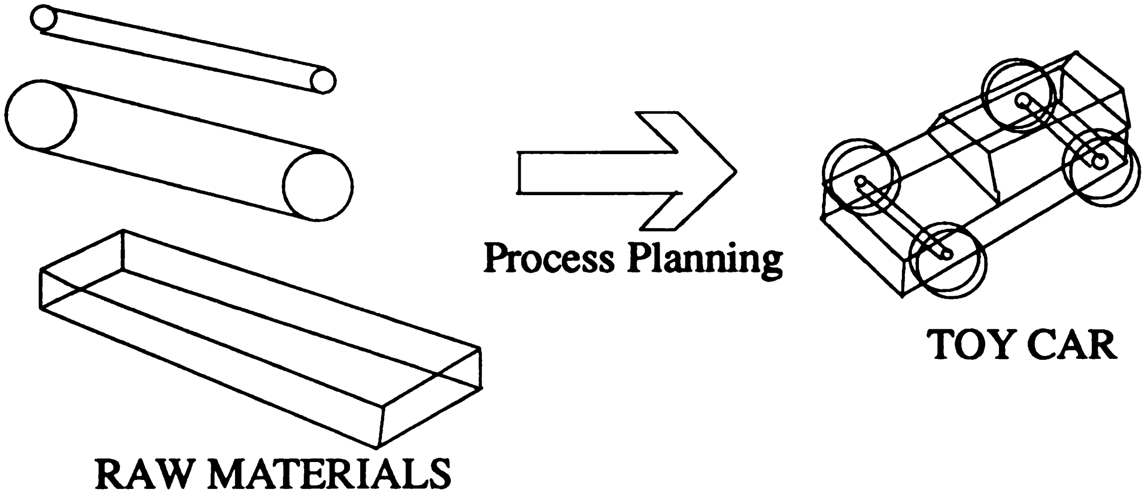
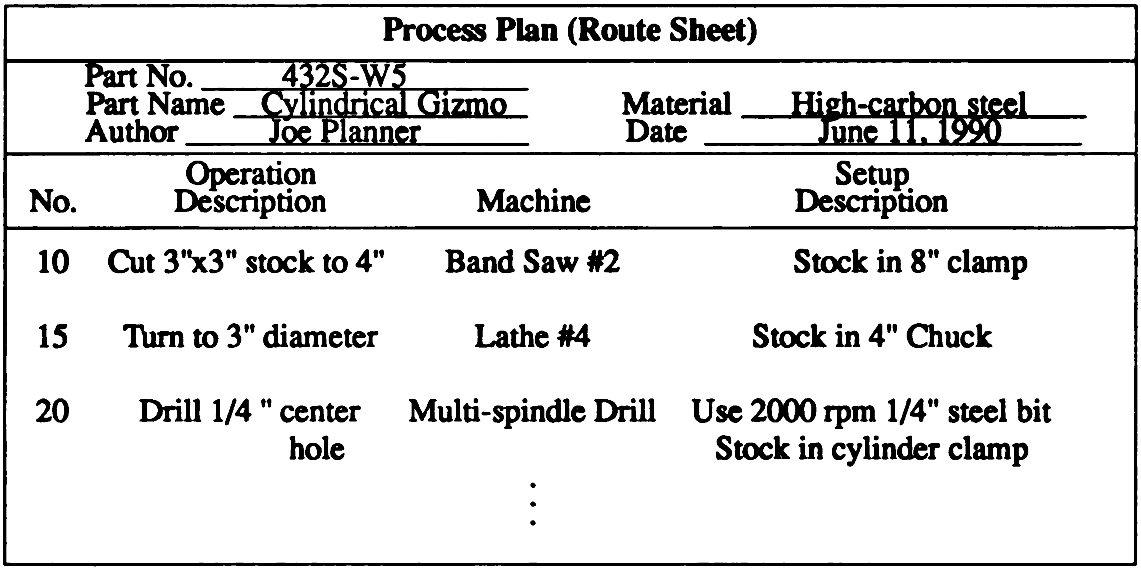


Figure 1-1: Raw Materials → Final Product

In general process planning is a complex activity and there is no guarantee of an *optimal* manufacturing plan being produced. An *optimal* process plan is least costly in terms of the time or money required to produce the desired part. A goal of computer-aided process planning is to produce *optimal* process plans automatically, given a product design. However, compromises are made to allow a small amount of human intervention, and suboptimal but usable process plans often result. Computers are being used with increasing frequency in this domain to maintain consistency in the detail and quality of process plans and to optimize the resulting manufacturing plans at either an individual product level or global factory level.

Process planning is one stage in Computer Integrated Engineering (CIE). CIE is the use of CAD/CAM technology to integrate the engineering functions of *product design, specification*, and *production* [Rich86]. Computer-aided process planning can actually help bridge the traditional gap between Computer Aided Design (CAD) and Computer Aided Manufacturing (CAM). Not only is the process planning problem of important practical interest to CIE, but it is of theoretical interest in computer science because it requires both ordinary computer power and high-level heuristic decision-making. The task of process planning is complex; no general purpose automatic process planners have been built. Even human experts are limited to particular domains by the large amount of knowledge and experience needed.



Process Plan (Route Sheet)

Part No. 432S-W5
Part Name Cylindrical Gizmo   Material High-carbon steel
Author Joe Planner   Date June 11, 1990

No.	Operation Description	Machine	Setup Description
10	Cut 3"x3" stock to 4"	Band Saw #2	Stock in 8" clamp
15	Turn to 3" diameter	Lathe #4	Stock in 4" Chuck
20	Drill 1/4 " center hole	Multi-spindle Drill	Use 2000 rpm 1/4" steel bit Stock in cylinder clamp
		⋮	

Figure 1-2: Example of a Process Plan (Route Sheet).

A concrete goal of Computer-Aided Process Planning is to automate as much as possible the creation of a *process plan* or *route sheet* from the description of a product. For example, in process planning for discrete metal parts, a *route sheet* or *process plan* gives the detailed routing or processing steps that are applied to a set of raw materials to create the desired product. More specifically, it contains information on the machines, tools, speeds, feeds, and other machining information to be used in changing the raw materials into the final product. Figure 1-2 shows an example process plan for a cylindrical part.

Process planning bridges the gap between the product design and manufacturing stages. Automation of the process planning step is important for several reasons:

- More human process planners are required in the U.S. than are currently available [Chan85].
- The routing sequences can be standardized for similar parts.
- More *efficient* (less costly) process plans can be produced.
- The productivity of process planners can be increased.
- Other application programs and simulation steps can be incorporated into the process planning task [Groo80].

Automating the process planning stage and integrating it with the other manufacturing stages helps to reduce costs and increase productivity.

Standardization of process plans through automation can minimize the number of different process plans for similar parts. Situations where many suboptimal process plans are in service occur more often in a factory environment without the centralizing effects of automation. For example, in a factory setting, a total of 42 different routings, covering 20 different machines, had been developed for various sizes of a relatively simple part called an 'expander sleeve.' The reason given for the large number of routings was that eight or nine manufacturing engineers, two planners, and 25 NC part programmers had made the 42 nonstandard plans. "Upon analysis, it was determined that only two different routings through four machines were needed to process the 64 different part numbers" [Groo80]. *Group technology* is a classification and coding system for identifying similar parts by their code numbers and is used to index existing plans. It can help avoid duplication of effort and use of less efficient routings when used in computer-aided process planning to identify similar types of parts.

1.2. The Manufacturing Cycle

An overview of the manufacturing cycle is beneficial to see how process planning fits in and how it can be used to improve the manufacturing cycle. Generally the following groups are active in the manufacturing cycle [Groo80]:

1) sales and marketing
2) product design and engineering
3) manufacturing engineering/process planning
4) industrial engineering
5) production planning and control
6) manufacturing
7) quality control
8) shipping and inventory control

The initial call to make a product comes from the sales and marketing department. This is the result of marketing predictions or the requests of customers. The product design and engineering group will use the part specifications and requirements to come up with a detailed design for new products. Feedback from the other groups on the ease of manufacturing, maintenance, and cost helps determine a viable design. By automating the process planning task it is possible to take the specifications of a design and to quickly provide the design and engineering groups with a set of projected production times and costs for the product. This feedback allows the designers to improve the design for manufacturability and profitability.

Manufacturing engineers are responsible for creating the process plan for the designed part. The tools and fixtures needed to manufacture the part are also specified at this time. The industrial engineers determine the work methods and time standards for the individual production operations [Groo80]. Production planning and control formulate a master schedule of quantities and delivery dates for the products being manufactured in the factory. The master schedule is used in material requirements planning (MRP) to determine the raw materials needed for the factory to produce goods. The other main task of the production planning and control group is production scheduling and maintaining the schedules despite dynamic problems on the shop floor.

The manufacturing group produces the final products from the initial raw materials. Quality control is responsible for checking the quality of the product as it is transformed from raw materials into the final product. The final step in the manufacturing cycle for a product involves the maintenance of inventory levels and shipping schedules to meet customer demands. For the interested reader the book by Chang and Wysk [Chan85] is a good introduction to CAPP systems with additional emphasis on the manufacturing, design, and process engineering functions. Concurrent design of products and processes is discussed by Nevins and Whitney [Nevi89].

1.3. Approaches to CAPP

Currently, two main approaches are used in CAPP [Phil85]. The *variant* approach has a library of stored process plans that have been used in the past. This database of stored parts and process plans is searched to find a process plan for a part that is of a similar type and physical geometry. Ideally, the process plan for the similar part will be close to a good process plan for the new part.

In contrast to the *variant* method is the *generative* approach. Old or similar process plans are not used in the generative approach. The *generative* approach creates plausible process plans based on knowledge of the machines in a factory, the raw materials available, and knowledge of which machines can create certain features. Each generated plan is evaluated according to a cost function so that an optimal solution or set of the most efficient process plans can be identified.

1.3.1. The Variant Method of CAPP

In variant process planning the process plan for a similar part is used as a template to create the process plan for a new part. Typically, an experienced user is needed to help the computer form a process plan from this template. The most popular variant systems are CAPP, Miplan, and AUTOPLAN [Bere86].

A common misconception is that most parts are made in large manufacturing batches. However, the Department of Labor estimates that 75% of all metalworking in the U.S. is done in lot sizes of 50 or less [Lind83]. Thus, a variety of different parts are made in the typical job shop. An important aspect of variant planning is the use of group technology codes to categorize parts into groups. This can help avoid creating process plans from scratch for parts that are only slightly different from parts in the database. Parts are usually grouped by physical geometry, production features, or from production flow analysis to group parts with similar operation sequences. Different physical and/or production features are encoded into a digital code that describes relationships and similarities among parts in the database. The code can be *hierarchical* with successive digits having meaning dependent on the preceding digits, or it can be a *chain code* with each digit representing a given feature independent of the other digits. A combination of these two schemes is generally more flexible and can be used by all groups involved in the manufacturing cycle. The purpose of a code is to serve as an index and grouping function for a database of parts.

Many coding and classification schemes have been developed to suit individual company and machine shop needs. A few of the more popular systems include the Opitz, CODE, and MICLASS systems [Lind83]. All three of these codes use both hierarchical and chain coding components to categorize parts. The basic Opitz coding system consists of nine digits which hold both physical geometry and manufacturing data. These digits hold information as follows:

Digit	Description
1	rotational/nonrotational part classes
2	main shape
3	rotational machining
4	plane surface machining
5	additional holes, teeth and forming
6	dimensions
7	material
8	original shape of raw material
9	accuracy

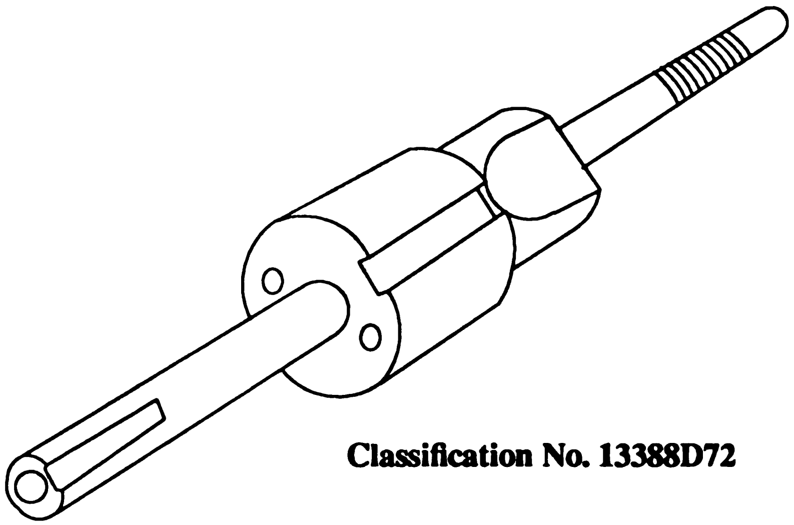


Figure 1-3: Rotationally Symmetric Part [Lind83]

In addition to these 9 digits another 4 digits may be added for particular applications and job shop dependencies.

The CODE system consists of eight digits with each digit taking on one of the 16 hexadecimal code values. For example, the part shown in Figure 1-3 is encoded as 13388D72.

The eight digit code 13388D72 is broken down as follows:

Digit	Description : Value
1	basic shape : cylinder = 1
2	concentric parts or parts diameters : multiconvex cylinder = 3
3	center hole type : a single blind hole = 3
4	non-center holes : bolt circle (2 holes) = 8
5	grooves or outside threads : threaded on one end = 8
6	combination of concentric variations, flats, slots, and protrusion features : sum of concentric variations (1), slots (4), and flats (8) = 13 (D)
7	major outside diameter : falls in range of 1.2" - 2.0" = 7
8	overall length of 1.5" : falls in range of 1.0" - 1.6" = 2

MICLASS has 12 digits in its universal code with up to 18 additional digits available to tailor the code to the individual company. MICLASS is set up so that the code for a part can be generated by a computer that asks the user a series of 8 to 20 questions about the part. The work part attributes coded in the first 12 digits of MICLASS are as follows [Groo80]:

Digit	Description
1	main shape
2 and 3	shape elements
4	position of shape elements
5 and 6	main dimensions
7	dimension ratio
8	auxiliary dimension
9 and 10	tolerance codes
11 and 12	material codes

These codes are used not only in retrieving and storing similar parts and process plans in a database, but also by many other users in the CIM environment.

1.3.2. The Generative Method of CAPP

Generative process planning uses machining and tooling knowledge along with properties of the raw materials to determine candidate process plans. Each generated process plan is evaluated to find its cost to choose the best process plans. Numerous combinations of machining operations may be considered and the most appropriate operations chosen using heuristics. Variations of promising formats are tried in hopes of finding an optimal solution. Heuristic cost functions play a key role in narrowing the large range of possible plans to a workable set of good possibilities.

A few of the generative systems are CADCAM, Acaps, CIMS/PRO, CPPP, Autap, and DCLASS [Chan85, Rich86]. Berenji has designated a third class of process planners: *Artificial Intelligence based* process planners, which are actually a subset of the generative class [Bere86]. They use AI techniques and the encoded knowledge of human expert process planners to generate the final process plans. This class includes the planners: Gari, Tom, PROPLAN, TOLTEC, and Hi-Mapp [Desc81, Mats82, Phil85, Tsat87, Bere86]. Generative systems do not require the use of group technology; however, the benefits of group technology to many other production engineering functions make it an essential element in the manufacturing environment [Berr84].

Proofs of optimality exist for the generative approach only when it is used in a restricted "ideal" search space. In realistic situations no such guarantee exists for either the variant or generative approach. If the problem space can be set up in such a way as to facilitate an optimal search algorithm on an ideal search space, such as A^*, gradient descent or alpha-beta search, then it may be possible to find an optimal process plan for a particular criterion function [Nils80].

Some tools and representation schemes from the AI domain can be used to aid the process planning task. For example, AND/OR graphs, inheritance properties, and frames have been used in some CAPP systems. Similarly, many other AI techniques are useful in attacking the process planning problem. One of the most prevalent problems in automatic process planning is that of large search spaces. The use of knowledge or heuristics about the problem domain is one way that difficult search problems have been handled in the past. Expert system shells are often used in creating process planners to handle the large amount of domain-specific knowledge. These topics are discussed in Chapter 2.

1.3.3. Assembly Planning

Assembly planning is an integral part of process planning. Products with more than one subpart must be assembled. The assembly, machining, and processing steps may be intermixed for some products. The sequencing of the these steps is very important in creating an efficient process plan. Assembly planning requires a computer representation of the components to be assembled. This information may be in the form of CAD data and/or a high-level representation (such as a part mating graph) of the components and their connective relationships. Often geometric reasoning techniques are used along with CAD data to determine valid assembly trajectories for subparts.

The general task of assembly planning involves determining not only the order of assembly for a set of n parts, but also finding valid grip points, dealing with the reality of gravity, part tolerances, and path planning. Optimal path planning in three-dimensions, even among polyhedral objects, is itself a problem which is known to require significant resources [Shar86]. This is only one of many challenging problems that CAPP presents to computer science. Chapter 3 discusses assembly planning in more detail.

1.4. The Organization of the Thesis

The remainder of this dissertation and its contributions are outlined below:

Chapter 2 surveys different types of CAPP systems. These examples illustrate some of the representational and computational difficulties involved in creating process plans automatically. The advantages and disadvantages associated with a spectrum of 3-D representation schemes are given along with other useful representations such as IF-THEN rules, frames, precedence graphs, and AND/OR graph representations. More intelligent and more powerful feature based geometric modelers and expert systems are necessary before CAPP can take an important place in the overall manufacturing cycle. It is shown that brute force search over alternative operations is intractable, and that heuristics such as scripts are helpful in organizing common subsequences of operations [Nau86]. The effects of gravity, fixturing considerations, and the necessity of avoiding unwanted collisions between machines and materials stretch our existing knowledge of common-sense and geometric reasoning.

Chapter 3 addresses the lack of research dealing with individual fasteners in the automated assembly planning literature. The representation of fasteners and planning for fastener application is of critical importance in creating optimal assembly plans. Additionally, most assembly planning systems require supplemental information on the relationships between various subparts. These issues are addressed by the Automated Assembly Planning with Fasteners (AAPF) system. AAPF accommodates fasteners and automatically determines feasible disassembly sequences based on the geometric data of the product. Under the assumptions of rigid subparts and no interior nor exterior forces, the reverse of a disassembly sequence is a valid assembly sequence. A mechanical mouse can be assembled in over 58 million different ways. Finding an optimal solution is not an easy task. Many different orderings exist for the set of tasks to be performed; some are much more desirable than others. Using a more practical assembly evaluation function than the heuristics and assumptions made by the AAPF system shows that although AAPF can produce good assembly sequences for some products, its lack of knowledge about fixturing constraints caused the mechanical mouse assembly solution to require 50% more time than an efficient assembly sequence that accounted for fixturing constraints. Other process planning examples of good and bad process plans show an execution time difference of up to 100%, indicating the value of being able to find the efficient process plans in the large set of possible plans.

Chapter 4 investigates the number of different sequential orderings for a set of tasks to be performed subject to a set of precedence relations. In machining, assembly, and general task planning, it is beneficial to know how many different task orderings are possible so that an efficient solution can be found in the most appropriate way. Although small search spaces can be enumerated, larger search spaces require heuristics and search techniques to limit the number of solutions evaluated to find a good solution in a timely manner. In general, determining the number of orderings for a set of tasks and constraints is intractable. To facilitate solving this problem, precedence graphs are used to represent the tasks and constraints. An *Oracle* graph reduction algorithm is presented that can quickly handle a class of graphs called *N-fold* graphs. *N-fold* graphs can be evaluated precisely using the three subgraph reduction patterns defined in this chapter. Arguments on the correctness for each of the three subgraph formulas are provided. After giving a set of precedence graph examples, several techniques are described for increasing the number of graphs that can be handled precisely.

Chapter 5 characterizes the *Oracle* algorithm and reveals implementation details. The uniqueness properties of the three subgraph replacement patterns and that of the final result for the number of possible orders for *N-fold* graphs is shown. A derivation of the time and space complexity [Horo89] for each part of the *Oracle* algorithm concludes that $O(V^2 * E)$ time and $O(V^2)$ space are required. The lower and upper bound estimating procedures are outlined for non-N-fold graphs and proofs of correctness are given. Empirical results on the percentage of graphs that the Oracle algorithm can handle exactly show that the algorithm can exactly evaluate most graphs with either a small or very large number of edges. Both an upper and lower bound on the number of possible orders is returned for graphs that cannot be evaluated exactly. Simulation results for the Oracle algorithm on random graphs show an $O(V^2)$ time complexity for the range of graphs tested.

Chapter 6 describes a CAPP software package for Concurrent Design and Evaluation (CDE) of products. Many of the components for this software package have been implemented here at M.S.U. or at other universities and industries. A single integrated package is described, but not implemented, which will aid in the design, manufacturing, and process planning tasks. Areas for future research in creating the proposed CDE system are discussed.

Chapter 7 concludes with a discussion of the contributions of this thesis, answered and unanswered questions, and areas for future work.

Chapter 2

Computer Aided Process Planning

Several process planning systems are studied to illustrate some of the difficult problems that arise in CAPP. The variant CAPE system is useful for standardizing parts, fixtures, and routings in process plans. CAPE uses group technology to connect its different modules and functions together and to enable similar parts to be easily identified, stored, and accessed. A generative system called DCLASS is discussed that enabled a 3:1 increase in productivity for sheet metal process planning. Unfortunately, when the DCLASS system was expanded to handle simple rotational parts, the lack of a good representation scheme made the system difficult to use. In fact, the designer quickly became frustrated with the large number of questions that DCLASS asked about the rotational part being created.

Unlike the DCLASS implementation, an AI based process planner called PROPLAN performs well on a set of simple rotational parts. Using a CAD database, the system derives the information necessary to create process plans for rotational parts. Key to the success of the PROPLAN program is the representation used to transform the problem from a three-dimensional problem to a two-dimensional one. This points out the importance of finding an appropriate representation for the planning problem. The WoodMaster system uses constructive solid geometry (CSG) representations of wooden products and a knowledge base consisting of machining information and the factory environment to determine a process plan. Determining the feasibility of machining operations or optimizing a sequence of operations is a difficult task.

Assembly planning is needed whenever two or more subparts make up a product. A large body of literature addresses the assembly planning and designing for assembly problems. However, the field of automated assembly planning is still in its infancy. Determining efficient assembly sequences is difficult due to the three-dimensional nature of the assembly environment and the combinatorial explosion of the number of possible assembly sequences for N subparts.

As with many issues in computer science, the representations used in CAPP are extremely important. Significant problems are posed in the areas of knowledge representation, geometric modeling and reasoning, planning, combinatorics, and searching for efficient solutions. Often several solid modeling schemes are used in CAPP so that the most appropriate representation can be used for a given subproblem. The most common three-dimensional representation methods are discussed and then used to represent a cylindrical tube.

Manufacturing knowledge and expertise can often be efficiently represented using IF-THEN rules, frames, scripts, precedence graphs, and AND/OR graphs. The benefits and drawbacks of these representation schemes are discussed since they are currently used in many of the automated manufacturing and assembly planning systems. Many of these representation schemes are borrowed from computer science. Artificial intelligence, VLSI design, operating systems and scheduling theory all have benefited from and can contribute to CAPP.

2.1. Process Planning Systems

Examples of the variant, generative, and artificial intelligence approaches to process planning are investigated to illustrate different aspects of process planning. Complex computer processing, data extraction, decision, and representation issues arise in process planning even for simple parts. It is important to find good representations for object geometry, the factory environment, process planning expertise, and the intermediate results obtained from applying machining operations to raw materials.

2.1.1. A Variant Approach: CAPE

The Computer-Aided Production Engineering (CAPE) system developed by the Garrett Turbine Engine Company [Berr84] is a variant system based on group technology part codes. Any given part belongs to both a routing family and a drawing family. A *routing family* contains plans with similar machining sequences. When new technology or manufacturing methods are introduced into the plant, it is possible to change all process plans in a routing family at the same time. The *drawing family* contains parts with similar geometric properties. This information is useful in processing queries of similar parts. For example: the part in Figure 1-3 is identified with the code 13388D72. Similar parts can be found by querying the database with the value 13388D72 or by specifying a subset of these digits such as 13?88D72 to find similar parts without worrying about the type of center hole in the part (3rd digit).

Geometric details are stored in the Part Description Program (PDP), along with surface tolerances, finishes, and other non-geometric data. Great effort is placed on using standard fixtures, routing, and tooling wherever applicable to reduce set-up time, change-over time, and stock inventory. The power of CAPE lies in its modularity and expandability.

2.1.2. A Generative Approach: DCLASS

A number of CAPP systems use expert system technology to automate the task of process planning. DCLASS is a decision-making and classification system developed at Brigham Young University. It uses a hierarchical representation scheme to store the product domain, factory environment, and other process planning knowledge. The DCLASS system was first used to handle sheet metal parts, and later extended to handle a class of simple rotational parts. Even though working with sheet metal may produce a three-dimensional part, from an engineering point of view it can be thought of as a two-dimensional problem [Rich86]. DCLASS queries the user via menus about the part to be created. Information about the raw materials to be used for the part such as its shape, number of holes, slots, and other part features are obtained in this manner. Similarly, other process information (such as heat treatment or painting) is obtained to create a process plan. The knowledge base in DCLASS is contained in hierarchical decision trees that are used to represent logical groupings of information. The decision trees are used as production rules by a forward chaining inference engine to produce the process plan for the part.

After eight months of development, a system for sheet metal parts was tested in a manufacturing environment. DCLASS generated a process plan for a part and then allowed the human process planner to modify the plan to improve its quality. A productivity increase of 3:1 was realized in the sheet metal process planning effort itself [Rich86]. Another benefit of the generative process approach was an increase in the consistency of detail and correctness of the process plans. In several instances, design flaws or mistakes in the design specifications were discovered when the specifications called for an operation that didn't appear on the selection menu presented to the user. For example, since no painting of aluminum was performed in their job shop, such an option would not appear as a valid choice in the user's display menu, [Rich86]. Without the program the human process planner would have written the plan by hand. If the human process planners forgot that painting couldn't be done on aluminum, they would have created an improper process plan that may not have been noticed until it reached the factory floor.

The next step in the development of DCLASS was to create a system for a small subset of simple rotational parts. Unfortunately, the number of special features (such as traverse holes, bevels, chamfers, threads, countersinks, counterbores, spherical ends, tapers, and knurls) in such parts was large. So many questions were asked of the users that they became annoyed and frustrated. The number of parameters associated with more complex parts and their production made further attempts at generative process planning with DCLASS infeasible. If a CAD model were available, it may have been possible to derive the answers to many of the system's questions from the CAD data. Both the PROPLAN and WoodMaster systems described below work directly from the CAD data to free the designer from being asked for this information again. Another problem encountered was the development and maintenance of the complex decision logic used in DCLASS. Changes in the factory environment or in a class of parts that are addressed by the system must be reflected in the hierarchical decision structure. It takes time and careful thought to change the decision making logic in such a way as to ensure the continued validity of the resulting process plans.

2.1.3. An Expert System (AI) Approach: PROPLAN

Although DCLASS was found to be inadequate for simple rotational parts, the PROPLAN system proved to be less demanding on its users [Phil85]. PROPLAN is an example of a knowledge-based approach to generative process planning that creates process plans from CAD data to be used in a CAM environment. This system avoids the difficulties associated with asking the user for all the part specifications by deriving this information from a CAD database containing the part. PROPLAN is capable of analyzing the part geometry in its internal symbolic form to come up with a logical sequence of operations to be performed on some raw material to produce the final result. The resulting process plan specifies what machines and tools (drill bits, cutters, saw blades) should be used, what type of coolant is needed, and the speed, feed, and depth of cut for each operation. PROPLAN is also capable of explaining the line of reasoning it used to come up with the final process plan. This is helpful for developing and debugging the expert system and knowledge base. The system can be changed to accommodate different machinery and different types of tools.

INITIAL STATE: Material to be removed
or Area to be removed from grid

FINAL STATE: No material left
or Area in grid = NIL

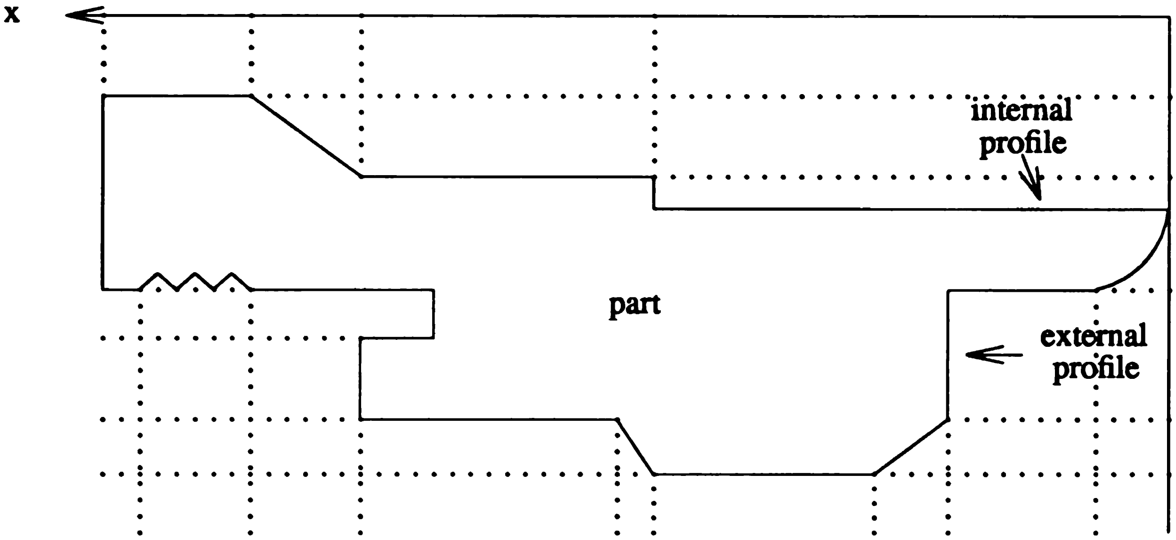


Figure 2-1: Rotational part representation used by PROPLAN [Phil85]

PROPLAN was implemented for a subset of rotational parts. Translation routines were used to convert part geometry from the CAD database representation to the internal symbolic representation used by PROPLAN. Part geometries were specified in terms of their internal and external profiles and additional features such as holes, slots, or keyways. Internally, this information was represented as a part tree consisting primarily of the locations of profiling lines and arcs. One problem with using a computer to create process plans is due to the computer's ineptness at organizing data by itself. From the representation of a part in a CAD database it is extremely difficult for the computer to know that a certain set of surfaces or lines represents (for example) a 1/4 inch hole that must be drilled. Therefore, a high-level understanding of the part geometry is required to create process plans. Higher-level part features such as holes, grooves, and adjacent surfaces must be represented in a format that can be easily manipulated and 'understood' by the decision logic in the expert system. (This is a strong motivation for the "feature-based" CAD systems under development today.)

PROPLAN represents the task of material removal as a two-dimensional graph that shows the area that must be removed to process the part. This area is then subdivided into fixed-size sectors, rectangles, and triangles by drawing vertical and horizontal lines at every point of the profile segments [Phil85]. The fixed size of the sectors is determined by the amount of material that can be removed in one pass by a specific machine and tool type. Operations such as turning a part on a lathe to remove a layer of material are viewed as removing a rectangle from the internal two-dimensional grid representation of the part. This simplification allows the three-dimensional problem to be viewed as a two-dimensional area removal process.

Process plans were developed for several parts by PROPLAN and then compared to the process plans currently being used in a factory environment. The plans were similar; however, some differed because of local constraints existing in the factory that were not accounted for in the expert system. Human process planners tend to describe operations at different levels of detail depending on how they feel -- for example, rushed, tired, etc. The computer-generated process plans had a more consistent level of detail than those from the human process planners. This is desirable at times to make sure the resulting product meets desired standards. The amount of detail is specified in the knowledge base of the expert system and could be increased or decreased as desired. The results did show that the system was efficient and had the potential for practical application [Phil85].

2.1.4. The WoodMaster Process Planner

The WoodMaster system successfully creates a process plan for wood objects made up of block and cylinder primitive objects requiring hole making, lathe, or cutting operations [Mill88]. An infinite number of possible objects can be defined using the CSG model even though the geometric primitives are simple. A more useful program could be created with the addition of code, primarily in the form of machine and stock specification tables, for other tools and stock materials. However, to create a viable industrial product more reasoning power is also required.

The time taken by WoodMaster to create these process plans is comparable to the time taken by a human expert. It required 8 1/2 minutes and 15 minutes of computer time to create process plans for the pencil holder and toy car designs respectively. WoodMaster was designed as a graphical demonstration of process plan creation.

WoodMaster first reads a part definition file. This file contains CSG data for block, sphere, cylinder, and tube-type object primitives as well as three types of fastener primitives. After the primitive object data is read in, it is entered into the I-DEAS 3-D object modeling system and displayed. WoodMaster then scans the table of primitives sequentially for objects. For each object the program determines a process plan for creating the basic primitive shape (block, cylinder, or tube) of the proper dimensions. WoodMaster finds a piece of stock material with the correct dimensions to create each subpart and then checks the tool specification database to find which tools can produce each feature in the subpart. Other process plan details such as assignment of specific machines to perform the tasks, routing of materials, and detailed tooling parameters are not created by WoodMaster.

A graphics window is used to display each part as it is being processed. All voids are checked to see if they intersect the object[1]. Intersecting voids correspond to material removal operations. Although many operations are used in a job shop (drilling, boring, milling, planing, reaming, etc.), only drilling, lathe work, and cutting operations are implemented in WoodMaster. To extend the number of available operations, additional code (in the form of data tables) corresponding to those operations can be added to WoodMaster. Once all voids are checked for intersection and processed, remaining unprocessed objects are processed.

[1] A void is a negative primitive implying the boolean difference operation.

In order to create a working process planner for wooden objects several simplifying assumptions were made.

1) WoodMaster assembles the objects in the order they appear in the input data to avoid assembly planning problems such as considering how to grip the parts to be assembled.

2) The effect of gravity upon partial assemblies is not considered.

3) Only cutting, hole making, and lathe operations are needed to produce the individual parts of the object.

Even with such a restricted world of objects, many of the hard processing and decision making problems that appear in more complicated object domains arise. Problems such as the feasibility of performing a given operation and optimization of operation order occur at varying levels in the process planning task.

2.2. Assembly Planning

Much work has been done since the early 1960's on analyzing the process of assembly and in assembly line balancing [Akag80, Reym87]. The general assembly planning problem is treated in [Brod87, Hutc90, Lieb77, Loza76, Nevi89, Popp90, Sand90, Woo87], and examples showing a large number of possible assembly sequences are given in Section 4.6 of this thesis. One of the pioneers in the areas of Automatic Assembly and Design for Assembly is Geoffrey Boothroyd. He has investigated the methods and economics of automated assembly [Boot82]. Designing products for assembly can reduce part counts by 35 to 40% with significant reduction in assembly time. For example, the Pro-Printer dot matrix printer by IBM was designed for ease of assembly. The Pro-Printer requires three minutes to assemble and has 40% fewer parts than its predecessor, which required 30 minutes to assemble.

In assembling the product there may be many possible orderings; some are very inefficient and only a few orders may be "optimal". The assembly problem requires modeling how to grip objects, along which directions the assembly motions will be, and determining whether or not parts will *intersect* during assembly. As mentioned earlier, in the toy car example, it is impossible to place both wheels on the axle and then try to put this assembly into the car body. This fact is hard to determine without simulating the assembly motions and seeing that a collision between a wheel or the axle and the car body will occur with this assembly order. For human process planners, some of this is common sense, but for computers it requires geometric reasoning and knowledge. AI researchers are finding out that common sense is difficult to impart to programs. Applying chunks of expert human knowledge to the process planning problem can constrain the total number of relevant operations [Tsat87].

The literature dealing with the automatic determination of assembly sequences has just recently begun to emerge. However, the roots leading to automatic assembly planning systems can be found in work done in the areas of geometric representation [Requ80, Requ82, Kemp87], planning [Ste81a, Ste81b, Laug86, Cama87, Dona87], and artificial intelligence [Rich83, Wins84].

hTe assembly planning process is often carried out in a hierarchical fashion due to its complexity. This hierarchy is often broken down into assembly-level planning, task-level planning, manipulator-level planning, and joint-level planning [Huan90]. Huang and Lee have created an Automatic Assembly Planning System that consists of five modules: world database, simulated world model, knowledge acquisition mechanism, planning knowledge base, and assembly planner. CAD objects are first entered as CSG primitives and then transformed into B-rep which is used for all subsequent geometric reasoning operations. Their system automatically generates "an assembly plan subject to the resource constraints of a given assembly cell" and is designed to improve "the flexibility and productivity of flexible manufacturing systems" [Huan90].

Research in geometric reasoning about gravity, stability, and reducing the computational burden associated with the assembly planning of free form B-rep objects is under way by Hoffman [Hof90a, Hof90b]. In certain industries, such as the aerospace industry, B-rep formats (which are needed for their aerodynamic properties) are essential because they allow free-form surfaces to be used. The BRAEN automatic assembly planning system is one of the few systems to reason about the effect of gravity and stability of assembly using purely geometric information [Hof90a]. Due to the complexity of the stability problem, the approximation methods used by Hoffman are sufficient but not necessary for detecting instability of subparts. Heuristics are used to guide the search for promising removal directions and distances. Chapter 3 investigates automatic assembly planning with fasteners in more detail.

2.3. Representations for Process Planning

As the previous discussion on the expert system planner PROPLAN pointed out, the representation used can often turn a difficult problem into one that can be tackled by an automatic process planner. This section reviews general methods of representation that can be used in process planners. After discussing the importance of representation in solid modeling, the subsequent sections discuss IF-THEN rules, frames, and the use of precedence and AND/OR graphs in the domain of process planning. Some similarities between the computer science disciplines of artificial intelligence, operating systems, and scheduling theory are made to point out how process planning can benefit from the theory and representations used in computer science.

2.3.1. Importance of Solid Modeling and Representation

Central to the idea of process planning is the representation of real world objects inside computers. This is necessary in order for the expert system to know what three-dimensional object the process plan should create and assemble. A useful representation scheme should have a way to determine if a certain representation is *invalid*. A representation is *invalid* if it refers to an object that is not three-dimensional. For example, a straight line with no thickness (diameter) associated with it is an invalid three-dimensional object. Useful representations are also *unambiguous*. An *ambiguous* representation refers to more than one solid. Another desirable, although unnecessary property of a representation, is that there be only one way to represent a given solid. This is referred to as *representational uniqueness*.

There are many different schemes for representing solid objects. In the following representation schemes a figure shows how a tube would be represented. Some of the unambiguous schemes are: spatial occupancy enumeration, constructive solid geometry, swept volumes, and boundary representations [Requ80].

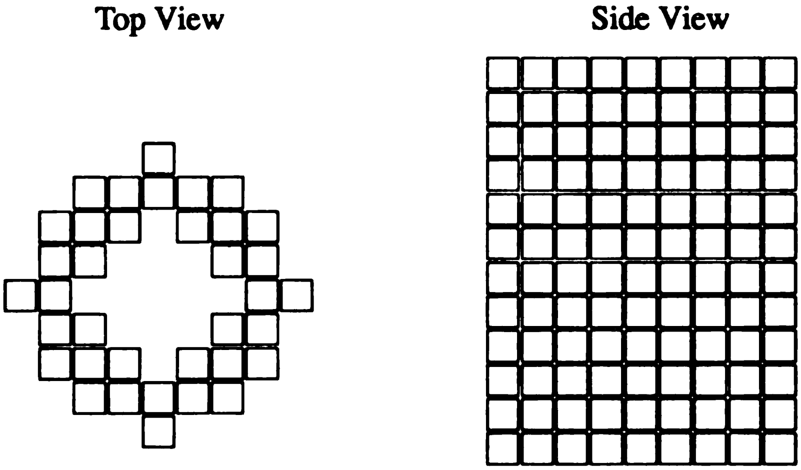


Figure 2-2: Spatial Occupancy Enumeration for a Tube

1) **Spatial Occupancy Enumeration** - Specifies an object by a set of volume elements located at fixed grid points. The representation is unambiguous, unique, and easy to validate. However, for describing parts to be machined this representation is verbose, difficult to use to precisely model curved objects, and difficult to manipulate because of the many primitives. The representation of a tube shape (below) shows that curved surfaces are approximated as closely as the cube-shaped elements allow. To get a better

approximation smaller volume elements would have to be used. This can lead to bulky definitions for objects.

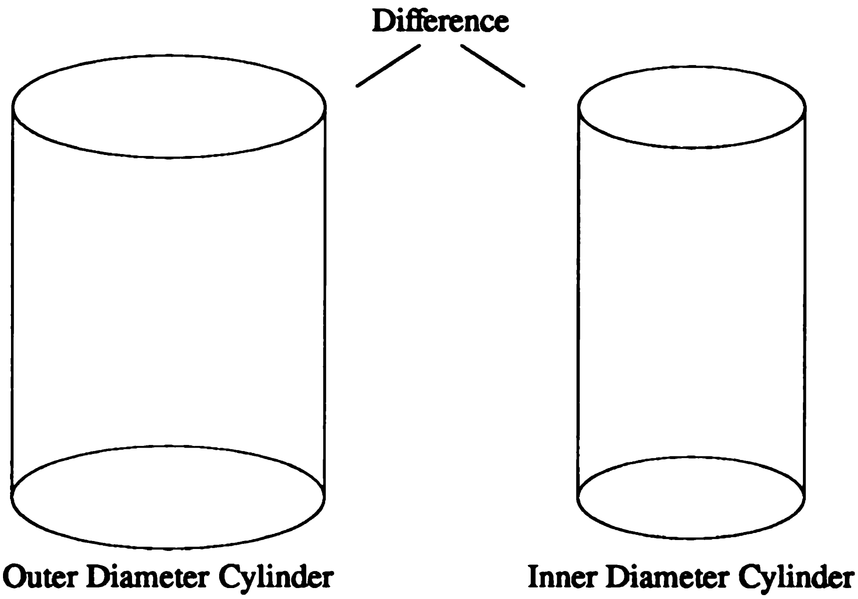


Figure 2-3: Constructive Solid Geometry

2) **Constructive Solid Geometry** (CSG) - Solids are represented as an ordered binary tree of Boolean constructions or combinations of solid components via regularized set operations. This scheme is unambiguous, nonunique, and generally easy for both creation and validation of object representations. CSG is one of the more promising representations for mechanical parts because it can easily represent most regular parts, has syntactically guaranteed validity, and is easy to use.

3) **Swept Volumes** - Specifies an object in terms of a cross sectional area that is translated through space along a specified vector. The volume swept out by the cross sectional area defines the three-dimensional object. This representation is useful in collision detection and material removal operations. For example, in machining a part, the material removed is the intersection of the volume swept out by the cutting tool, intersected with the volume of the part.

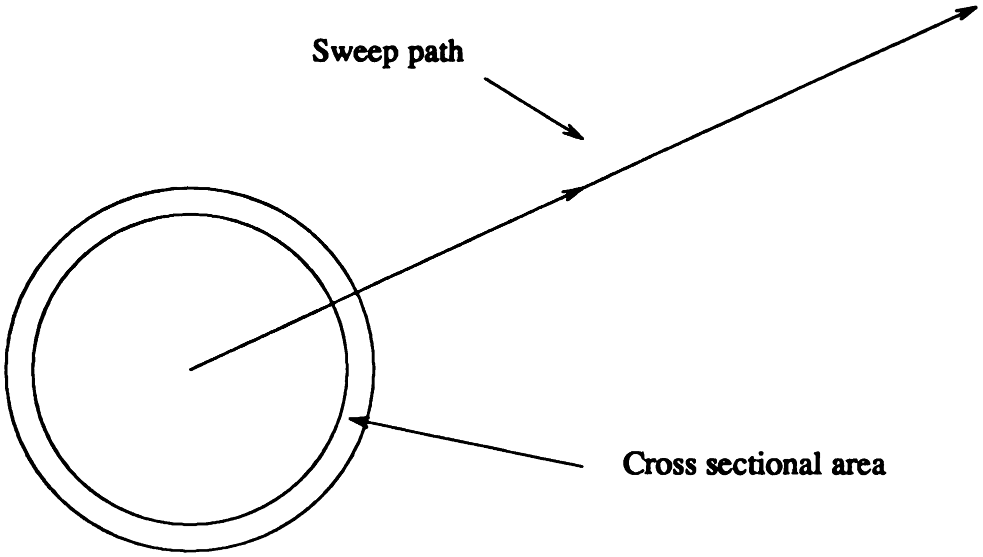


Figure 2-4: Volume swept by an area

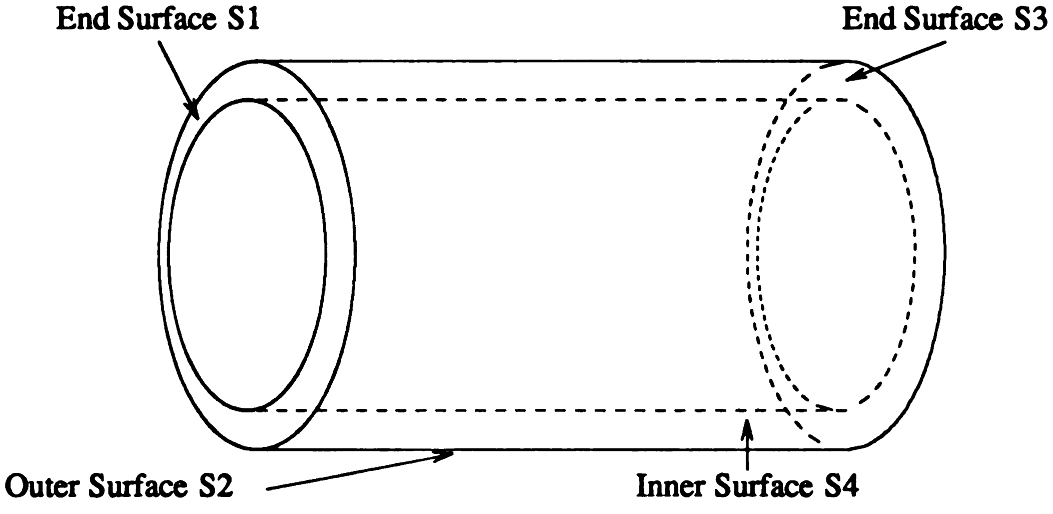


Figure 2-5: Boundary Representation

4) **Boundary Representations** - A set of faces (surfaces) bounding the occupied volume are specified. Each face is defined by a set of bounding edges and vertices or by using another surface representation, for example, plane, sphere, B-spline patch, etc. Boundary representation schemes are unambiguous if faces are represented unambiguously, but generally are not unique [Requ80]. Validity is computationally hard to deter-

mine. However, boundary representations are useful in displaying different views of a part and in representing the relationship between different surfaces and edges. Since they can be derived from CSG representations (which are easy to validate, etc.), they can be used as an aid to a CSG representation scheme.

The most commonly used geometric representations are CSG and B-rep. Both have a potentially large domain. CSG has syntactically guaranteed validity as well as being concise and easy to create [Requ80]. CSG and boundary representations are used in PADL2 to represent any solid bounded by any combination of arbitrarily oriented cylindrical, planar, spherical, and conical faces. To represent an even wider range of unsculptured and sculptured parts, several representation schemes could be used. However, changing from one representation to another can be very difficult to perform (if not impossible). Certain geometrical questions are more easily answered in one representation scheme than in another (and may even be impossible to answer in some representations). Thus, the use of one main representation scheme and several auxiliary representation schemes may be necessary to achieve a higher degree of versatility and efficiency.

2.3.2. IF-THEN Rules

Often expert human planners approach the task of process planning with knowledge chunks similar to IF-THEN rules. The following is a simple example in process planning [Brdy87]:

```
IF {hole to be made has a small tolerance value} THEN
        {1) Drill initial pilot hole on lathe,
         2) Ream hole to final radius and tolerance}
```

This is an oversimplification of how to drill holes with small tolerance values. Perhaps boring would be more cost and time efficient than reaming. Human process planners often explain their decisions in terms of more complex rules than this example, yet their rules follow the same basic format:

```
IF {SET OF PRECONDITIONS HOLD} THEN
        {PERFORM A SET OF TASKS AND OPERATIONS}
```

This type of representation can handle many problems and is the most frequently used building block in commercial expert systems today. Rules should only be used where feasible. Computational problems may result in deciding which rule to apply (fire) if many rules can apply to (are triggered by) a particular state of problem solving. If the task can be described by IF-THEN rules and the number of rules that must be active at one time is small, then the use of rules may be a good choice. Such production rules represent *compiled knowledge* from the human expert and can allow for efficient decision-making.

2.3.3. Frames

The natural way to approach a problem is not always through IF-THEN rules. Sometimes a hierarchical approach is a more appropriate way to view the problem. Hierarchical planners distinguish between important considerations and details [Ste81a, Ste81b]. Using a hierarchical approach can help reduce the exponential search problem associated with rule-based methods by refining down from high-level abstract states toward the details that bring about the desired abstract states.

Frames are a way to organize knowledge hierarchically. Frames are templates with slots associated with different characteristics of an object, action, etc. For example, a hole drilling frame could be set up as [Nau86]:

```
(def  twist_drill
      (operation  make_hole)
      (cost  1)
      (min_diameter  0.1)
      (max_diameter  2.0)
)
```

This could signify that a twist drill can be used to perform a make-hole operation, it costs 1 unit to perform, and can be used for holes ranging between 0.1 and 2.0 units in diameter. The operation value describes that twist drill is a sub-class of the make-hole class of operations, which itself may be represented by a superframe higher up in the hierarchy. Twist drill would inherit default values and other information from the make_hole definition. This frame, as well as others with an operation type of make_hole, would be considered whenever a hole was needed in the finished product.

In Patrick Winston's AI book, frames are used to aid in dealing with natural language understanding of news stories [Wins84]. They simplify the search task considerably. "Frames are important because they inform us about common situations, those involving common sense knowledge, by telling us what assumptions to make, what things to look for and what ways to look." [Wins84]. Frames organized into a hierarchy allow exploitation of relationships not explicitly available in single production rule based systems.

The usefulness of frames in the process planning domain has been shown [NaCh85, Nau86, Tsat87]. For example, a process planner for metal parts using removal operations written in Prolog is discussed in [NaCh85]. Unlike other frame-based reasoning systems where the problem-solving knowledge that manipulates the frames is made up of rules, frames are used in [NaCh85] to encode the problem-solving knowledge as well. This allows for the problem to be solved in a hierarchical manner with only the relevant frames being considered for application.

2.3.4. AND/OR Graphs

AND/OR graphs are often used in AI applications as goal trees and to store alternate plans. The goal is at the root of the graph and may have many levels of nodes below it. The levels alternate between AND levels and OR levels with all the nodes on one level being connected to the previous level with the same AND- (or OR-) type edge. This representation can be helpful in solving the process planning problem. AND/OR graphs can represent how goals and sub-goals fit together. This simplifies the processing of user queries on how goals were achieved and why they were attempted [Wins84, Nils80].

This representation could also be helpful in storing the final process plan where AND/OR graphs can be used to compactly represent all possible assembly sequences. Traditionally, a *fixed sequence* assembly ordering or *precedence graph* was used to hold the assembly sequencing information. A *fixed sequence* assembly order explicitly specifies a single assembly sequence, and a *precedence graph* represents several assembly orders implicitly by representing the constraints between pairs of objects that are to be assembled or between tasks to be performed. The AND/OR graph representation allows for efficient planning algorithms that can use alternative assembly sequences according to dynamic conditions in the factory environment. The scheduling efficiency of AND/OR graphs was found to be better than fixed sequence and precedence graph representations [Home86].

We now illustrate how AND/OR trees can be used to represent assembly sequences. Figure 2-6 shows the 35 possible assembly states of a toy car, which consists of four wheels, two axles, and a body. The car can be assembled in 184 different sequences if all seven parts are considered as unique. Although 35 states are shown, 22 states are *duplicate* states, and only 13 are distinct states, assuming symmetries are equivalent. For example, the second row in Figure 2-6 shows four different assemblies. However, since all tires are similar components, all four states are actually "permutations" of one state consisting of the car body, one axle with two wheels and one axle with one wheel. Only one unique assembly state is needed to represent these four states.

Figure 2-7 shows the 13 unique assembly states and the AND/OR graph connections that represent all the 14 unique ways to assemble the toy car. Arcs connect a state to its subassemblies that must be assembled to create that state. Two subassemblies are always assembled at once; this is denoted by two arcs sharing a common origin to designate them as AND operations that must be done to create the previous state. Whenever a choice of assembly sequences exists, different sets of arcs for each assembly sequence are used. For example, state S_6 in Figure 2-7 can be assembled two different ways.

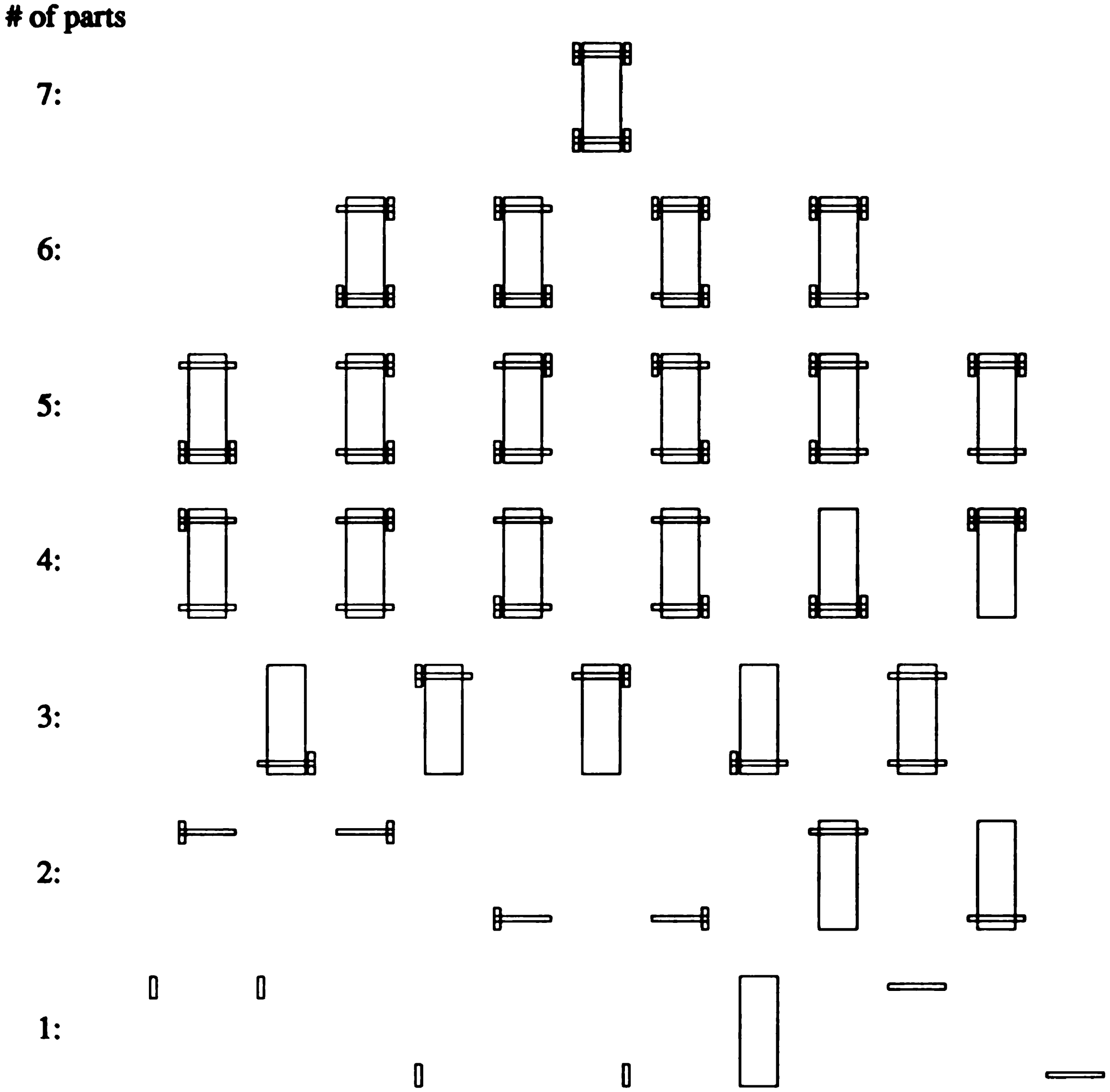


Figure 2-6: All 35 Possible Car Assembly States

Assembly operation a_1 or a_2 can be performed to create state S_6. Assembly operation a_2 creates S_6 by inserting the axle and wheel subassembly (state S_5) into the car body. State S_5 must be previously created by assembling an axle and wheel together, both of which are basic assembly units.

If this AND/OR representation is kept on-line in a manufacturing environment, it would be possible to assemble the toy car as quickly as possible dependent on the order

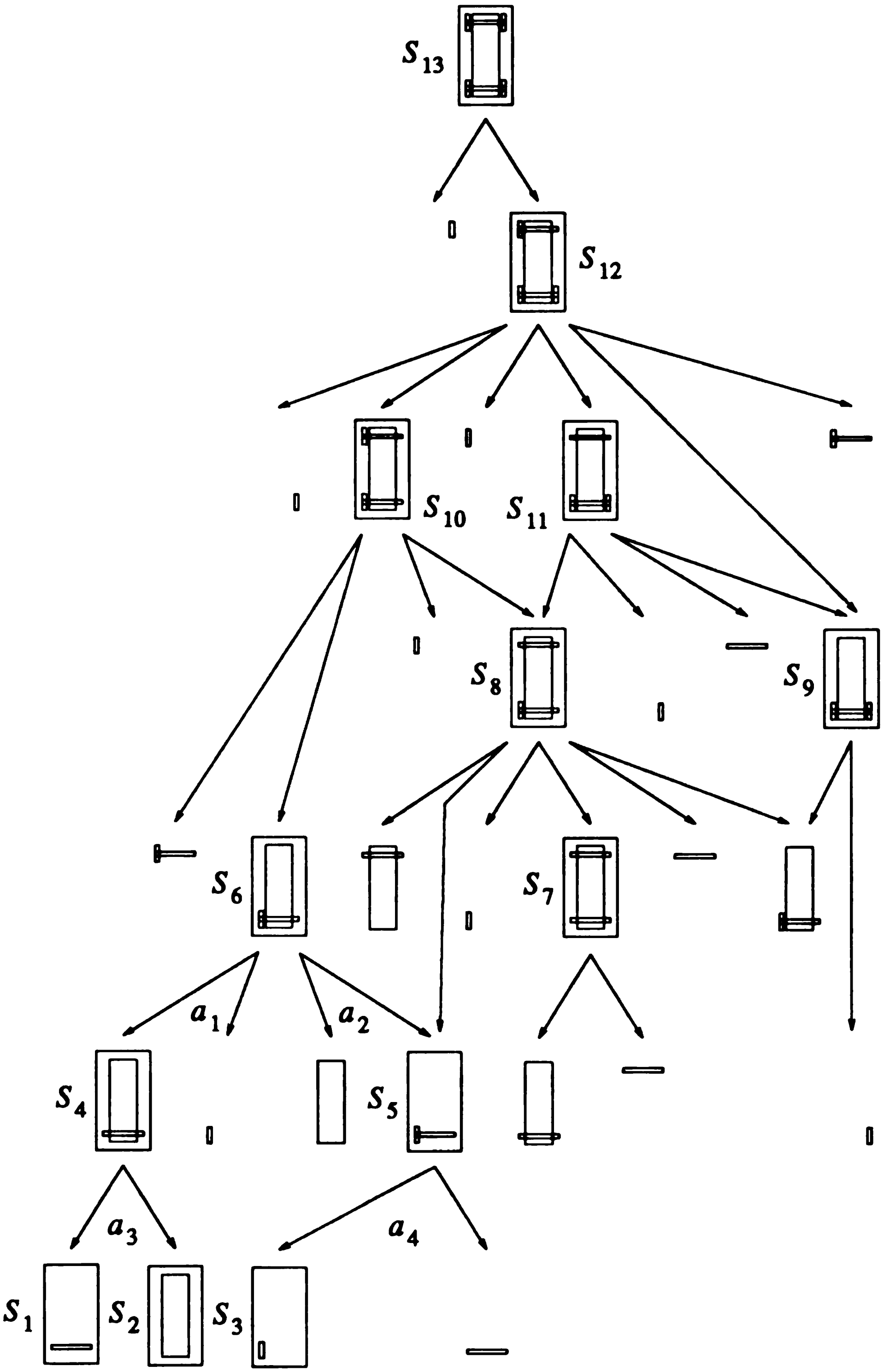


Figure 2-7: Toy Car AND/OR Assembly Graph

in which car parts or tools become available. Assuming a car body and axle were the first parts to become available, the system could start at the leaf nodes of the AND/OR tree representing the axle and car body to see if they shared a common parent state. In this example state S_4 could be created from the car body and axle. If the next object available were another axle, the system could see if state S_4 and an axle have common parents. They both have state S_7 as common parents, so the second axle could then be assembled with state S_4 to create state S_7. Continuing in this fashion allows parts to be assembled as they become available at a station instead of relying on a fixed assembly sequence. If no assembly sequence is possible, then no intersection of direct parents can be found, and the system must wait for another part.

A set of all the possible process plans stored this way has two main advantages over the sequential and precedence storage methods.

1) All process (assembly) plans are represented in one graph; a search for the least costly process plan can be done by searching this graph given a set of factory machines, raw materials, and machining costs. As new materials, manufacturing methods, or tools become available, the optimal process plan sequence can be found by using an AO* or similar search technique [Home86].

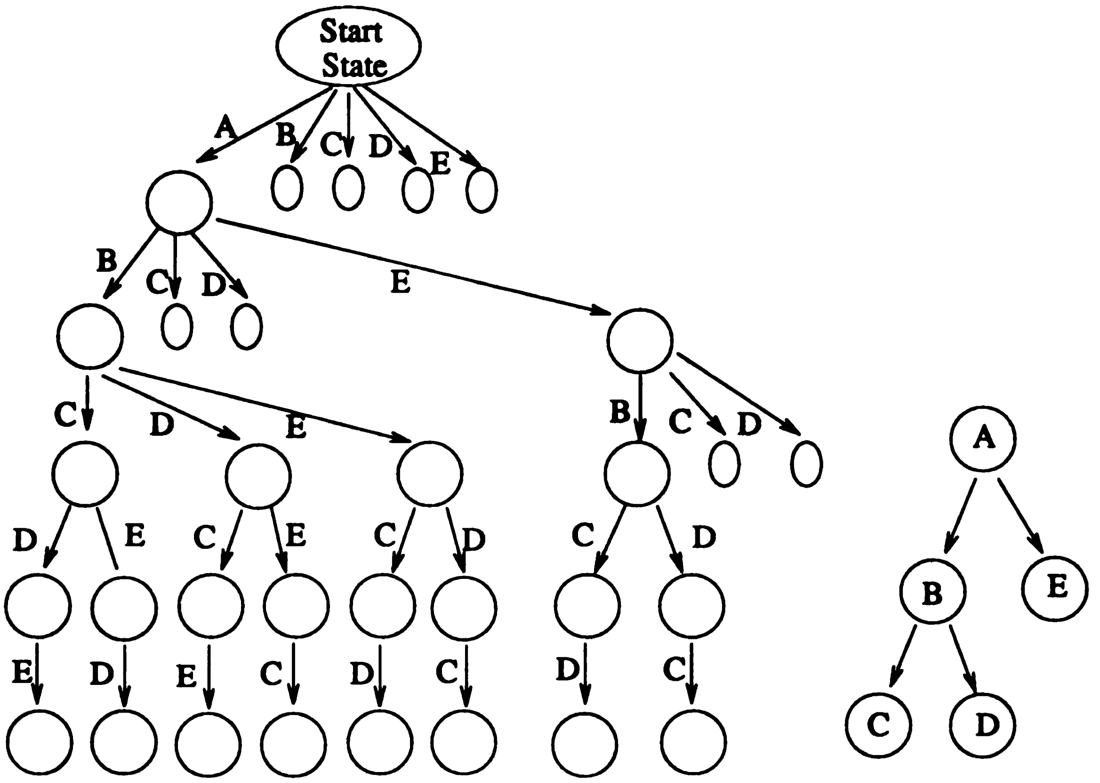
2) On the factory floor, scheduling and raw material delays often hold up operations. By having many or all process plans available, a computer could check to see if production could be continued using a different sequence with the available parts until the needed parts arrive. Keeping the process plan representation in an AND/OR graph format can also benefit flexible manufacturing workstation assembly. If parts arrive at varying times or are present in a bin with different parts overlapping each other, the system can select a valid assembly sequence (if one exists) for the parts that are most easily accessible. This can cut down on the quantity of parts that must be moved and stored, thus speeding up the assembly sequence.

2.3.5. Decision Trees and Precedence Graphs

In assembly planning the tasks to be performed consist of the steps necessary to join subassemblies together in creating the final product. Determining which tasks must be done before other tasks can be very time consuming. Given *l* connections (liaisons) to be made in an assembly, Bourjault outlined a method requiring at least $2(l^2 + l)$ yes or no questions to be answered by the user in order to determine the precedence constraints among assembly operations [Bour84]. Following this line of work, De Fazio and Whitney proposed a method where exactly $2*l$ questions are asked of the user. These questions require one of two responses: a) nothing restricts the assembly step, or b) a precedence relation between logical combinations or liaisons explicitly defining the precedence constraints for the given liaison [DeFa87].

Given N tasks to be performed with M constraints, it is possible to represent all valid orderings in a *decision tree*. Decision trees are referred to as Bourjault trees in the assembly plan representation survey by Wolter [Wolt90]. The vertices in a decision tree represent states, and the edges between vertices represent state transitions through the application of a particular task. For example: given a set of tasks {A, B, C, D, E} and the set of precedence constraints {A<B, A<E, B<C, B<D} (A < B indicates that task A must precede task B) the decision tree is shown in Figure 2-8 a. Every one of the 8 possible orderings of the four tasks are represented in this tree. In such a decision tree, determining a valid ordering of the tasks can be seen as a series of N decisions being made sequentially to decide which task is to be the 1st, 2nd, ... and last of the N tasks to be performed.

This sequence of decisions can be represented in a decision tree of height at most N levels. (Note: The precedence constraints are not explicitly represented in a decision tree.) The start state of the decision tree (level 0 of the tree) represents the starting configuration where no tasks have been performed. The 1st level of the tree, containing



a) Decision Tree

b) Precedence (AOV) Graph

Figure 2-8: Example of Decision Tree & Precedence Graph Sizes

children of the Start State, corresponds to the states where a single task has been applied to transform the start state. Depending on which edge is traversed (task applied) from the start state, different states are possible. Each child of a vertex at level k represents the resulting state following the application of a (k+1)th task being performed after the previous k tasks (k edges from the start state to a particular vertex at level k) have been performed. A path from the Start State vertex to a leaf at level N represents a valid sequential ordering of the tasks that does not violate any of the precedence constraints. The decision tree may represent from 0 to N ! possible sequences in different paths from the Start State to a leaf vertex at level N. Any terminal (leaf) vertex at a level less than N does not correspond to a valid assembly sequence because not all of the N tasks have been ordered at that point.

The toy car of Figure 2-7 would have a very large decision tree with 100,000 leaf vertices (paths). To avoid the problem of creating such an unwieldy representation it is possible to use an Activity On Vertex (AOV) graph representation. An AOV graph containing precedence constraints is commonly called a precedence graph. The tasks and constraints are represented in a precedence graph G = (V,E) containing N>0 vertices and M≥0 directed edges. Each task i ($1 \leq i \leq N$) is represented as a vertex V_i in the precedence graph. A precedent constraint between two tasks (say, i < j (i must precede j)), is represented as a directed arc from V_i to V_j in G [Horo76]. Figure 2-8 b shows the precedence graph for 5 tasks {A, B, C, D, E} and the precedence constraints {A<B, A<E, B<C, B<D}. Note that the different possible orderings are ***not explicit*** in the precedence graph as they are in the decision tree.

Although enumerative data structures such as the AND/OR graph and decision tree have many uses, they can be time consuming to generate and may also require large data structures to store all the possible assembly orders. The worst-case data structure size requirements for six enumerative data structures have been investigated by Wolter [Wolt90]. With 15 parts in an assembly the worst-case space requirements range from 10^7 for AND/OR graphs to 10^{20} for storing all possible state sequences. Like any enumerative representation that stores all possible orders, AND/OR graphs encounter the problem that all the task sequences must be enumerated. In the worst-case for 15 unconstrained parts the number of possible assembly orderings is $15\,! = 1.3 \times 10^{12}$.

2.4. Parallels to Disciplines in Computer Science

Automatic process planning is a new field and can benefit greatly by borrowing appropriate terminology and theory from other disciplines. Not only is there a wealth of information from the manufacturing environment, but much can also be learned from other disciplines such as computer science. For example, the areas of artificial intelligence, expert systems, natural language representation, operating systems, and scheduling theory have much to offer. Many theories and practical techniques have evolved in these areas as the computer developed.

2.4.1. Artificial Intelligence

Some of the tools from AI can be used to aid the process planning task. In the previous section the usefulness of frames, an AI representation, was considered for some problems. Similarly, many other AI techniques can be useful in attacking the process planning problem. One of the most prevalent problems in automatic process planning is that of large search spaces. An example is given in the next section to show that a large search space can result even for a simple part. The use of knowledge or heuristics about the problem domain is one way that exponential search problems have been handled in the past. This is one reason why expert system shells are often used in creating process planners. Expert systems are discussed in the next subsection with a few examples related to process planning.

In Section 2.3 we saw how artificial intelligence representation schemes such as rules, AND/OR graphs, and frames are useful in automatic process planning. Many algorithmic and heuristic tools have been developed in AI to aid in difficult search, understanding, and problem solving tasks.

Some of the more well known search techniques include gradient descent, minimax, alpha-beta, branch and bound, and A* [Wins84, Nils80]. These search techniques can benefit process planning when it is necessary to choose between many possible operations. The large search spaces in process planning need to be pruned down to consider only the most promising choices. As discussed in Section 2.4.4 scripts can be used to reduce the search space by categorizing knowledge and predicting the stages of various activities. Other planning systems and paradigms such as: RSTRIPS, DCOMP, amending plans, and hierarchical planning are applicable to the process planning problem [Nils80, Alte86]. Tsatsoulis and Kashyap provide a good history and summary of planning paradigms in relation to the process planning task [Tsat87].

2.4.2. Expert Systems and Process Planning

To fully automate process planning, it is necessary to capture the knowledge that enables the human process planners to come up with good, though not necessarily optimal, process plans. One way to do this is to incorporate the knowledge of human experts into sets of rules, frames, or nets in an expert system. Typically, the knowledge is gathered and put into an expert system in the five main stages listed below [Engl86].

1) Identify the problem and its difficulties.

2) Conceptualize the information of the experts into rules and guidelines.

3) Formalize the rules into the expert system representation.

4) Create the database of knowledge and appropriate control structures.

5) Test and evaluate the system performance.

Careful analysis of the human expert's thought process is critical. In addition to laws of nature, rules of thumb, and good planning practices, the expert system must be given common sense geometrical knowledge. The system must be able to 'recognize' that holes in a solid part can be made in a variety of ways (for example, drilled, bored, or cored.) Holes and other machining features can be used to direct a generative planner to find an appropriate solution. Machine tolerances must be matched with the desired finish of each surface in the end product. Cost functions for each of these processes must be accounted for so that the expert system can produce an economical process plan. Ideally, the expert system would produce optimal process plans.

Expert systems are computer programs which generally consist of four main modules as shown in Figure 2-9. The knowledge base is a set of rules and guidelines containing the human expert's knowledge about the problem domain. An example rule was given in Section 2.3.2. The global database contains only facts about the current

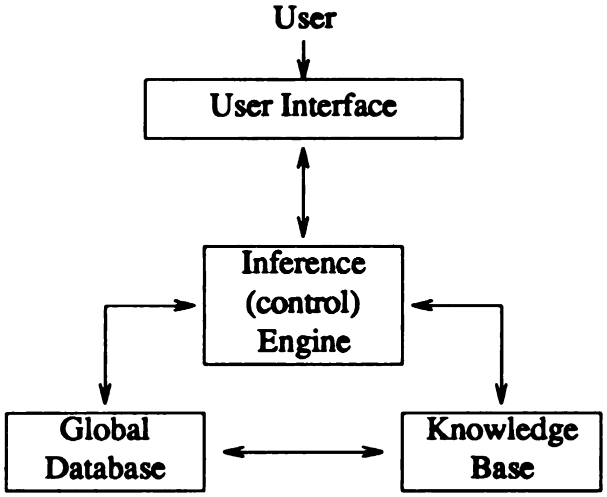


Figure 2-9: Information flow among the four modules in an expert system

case being considered and is updated on a case by case basis through the application of the rules. The inference engine is the control structure that determines when rules are checked and how the expert system arrives at its conclusion. The user interface handles data entry about particular cases and user queries to the database. Often the user interface includes a natural language interface that translates internal data and rule representations to a restricted subset of the English language. By keeping the four modules separate, it is easy to modify old rules or add new rules to the database without modifying the control structure. For many applications, it is possible to use a commercially available expert system shell containing an inference engine and user interface with an appropriate user-entered knowledge base (rules).

In an expert system the three-dimensional part model, relevant product features, surface tolerances and features, and the current state of the process plan as its being created are kept in the global database. The expert system uses facts in the global database to determine a valid process plan. Rules are stored in the knowledge base of the expert system to help determine a workable process plan. The inference engine simulates a human process planner's line of reasoning. A human process planner has considerable experience with the creation of process plans from the specifications of designers.

The designer may start out by determining the best raw materials to be used for major components of the product based on the functions to be performed by that component. The approach taken by the human process planner depends on the design specifications, quantity to be manufactured, intended use of the product, tools available in the factory, and physical geometry of the part. Analogously, the inference engine in the expert system determines which rules in the knowledge base to apply and decides if the rule conditions are satisfied according to various criteria.

In process planning, the expert system will be given a detailed part description and the goal of determining an ordered sequence of operations that can be applied to raw material(s) to produce the desired part. Optimization of the sequence of operations depends on where the various operations can be performed in the factory and how many operations can be performed without moving or refixturing the part. About 10-15% of the overall time required to produce a typical part is spent cutting metal. The rest is spent in setting up the part on the machine tool [Engl86].

Some operations must be done before others. For example a pilot hole should be drilled before using a large drill bit or performing a boring operation. In other cases it may be less costly to perform one operation before another. Even with the constraints placed on the ordering of the operations it may be possible to produce the same part in many different ways, so an efficient search algorithm is needed. Usually a tradeoff exists between optimality in the process plan and optimality in the search required to find the process plan. Knowledge encoded in the rules by the human expert can predetermine some sub-sequences, as discussed in the previous section, and can greatly reduce the search time.

The expert system should be able to 'explain' the reasoning behind its final choice. This is necessary during the development phase as an aid in debugging and is also useful for gaining user approval by letting the users know why it made certain choices. Another important facility desired in an expert system is the ability to easily update the database of machine types and tolerances as well as the decision-making logic. Expert systems in manufacturing design have been shown in a broader perspective in [Hera87, Rych88]. They give four problem areas in manufacturing design: part design, process planning, equipment selection, and facility layout.

The use of expert systems and AI techniques are well suited for the computational complexity and large search spaces associated with process planning. Knowledge bases and decision logic must be kept up to date in expert systems in order that process plans created by an expert system are viable in the real-world factory environment. Artificial intelligence languages and expert systems can play a key role in helping solve problems in manufacturing automation if appropriate representations can be found [Arda87].

2.4.3. Operating Systems and Scheduling Theory

Some similarities between operating systems and process planning are discussed below.

Resources and resource scheduling: In process planning, resources include the tools available in the factory, the raw stock materials, the machining methods, and the operators. In operating systems, resources include arithmetic and logical computational units, data in memory and on disk, printers, and the CPU. An operating system must provide for scheduling a group of tasks and the resources they need to optimize the flow of work through the system.

Transfer of parts/data: Aisle-ways, conveyor belts, and overhead lifts are used to move parts around inside the factory. Reducing of handling costs should be considered when an attempt at creating optimal process plans is made. This is another reason for keeping the process plan in an AND/OR graph until the item is made on the factory floor and the congestion of the various routing paths is known. In computer science, whenever computer architects design a computer system, they must provide for sufficient speeds on the data paths and for an appropriate number of paths. The operating system often applies scheduling and optimization techniques to use the available data bandwidths to the best advantage.

Optimization: In process planning the part to be manufactured must be made as cost effectively as possible given material, factory, and time constraints. The optimization and search techniques used can be borrowed from job scheduling, operations research, and artificial intelligence domains. Many theoretical results and scheduling algorithms have been developed for scheduling jobs in order to improve the throughput of computer systems. Additionally, if a large search space must be investigated to find an efficient solution, then an exhaustive search may not be feasible and an admissible or heuristic search technique from the domain of artificial intelligence may be appropriate. Although the A^* search algorithm is guaranteed to find an optimal solution if an admissible evaluation function is used, heuristic search techniques may not find an optimal solution.

2.4.4. Natural Language and VLSI Design

Scripts from natural language research can be beneficial to process planning by allowing a hybrid of the variant and generative approaches [Scha77, Tsat87]. A script is a sequence of actions or operations that are usually performed for some known stereotypic activity (for example: creating a 1/2" hole). When a high tolerance hole exists in the product being planned, scripts holding typical sequences such as *(drill, ream)*, *(drill, bore)*, or *(core, ream)* could be compared to find the best sequence. This avoids investigating and re-evaluating each of the many possible sequences for producing high tolerance holes. By using scripts it is possible to avoid searching all possible operations when it is known that after certain rough drilling operations a reaming operation is usually done to improve the tolerance of a hole.

In WoodMaster, for example, the part type 'tube' is defined to be a cylindrical piece of stock with a hole along its axis of rotation. The tube object type is similar to a script in that it combines the definition of a cylinder solid with a hole making process that creates a concentric hole. This saves the creation of another material removal (void) primitive in the input data as well as simplifying the search procedure.

In the TOLTEC CAPP system [Tsat87], memorized case-based sequences are used to create process plans. Using 180 knowledge-packages the TOLTEC system is able to plan the manufacturing of holes, threads, and turned surfaces. By taking advantage of the fact that each manufacturing step can only be performed in a few ways, the problem of an exponential search space is avoided[2].

Strong similarities exist between process planning and Very Large Scale Integration (VLSI) chip design and layout. Many circuit design and verification systems exist. For example, a hierarchical rule-based approach has been used successfully to automate CMOS circuit design and specification [Wu86]. However, general process planning is more complex than VLSI design. It differs in materials selection, available manufacturing processes, and its use of 3-D geometry [Dixo86]. Chip design is limited to a subset of possibilities in each of these areas.

One of the most difficult problems in CAPP is geometric reasoning about objects, machines, and tools in the manufacturing environment. For example, there are infinitely many ways that a 3-D solid object can be defined using Constructive Solid Geometry (CSG). Unfortunately, different geometric representations are more or less useful depending on the task to be performed, and it is difficult to decide which representation(s) to use. For example, displaying the edges of a three-dimensional model is an easy task using a boundary representation, but determining the intersection of two parts is more difficult in a boundary representation than a constructive solid geometry representation [Kemp87, Requ82]. Often several geometric representations are used to

[2] Scripts are also used in computer chess to avoid the large search spaces found at key points in a game of chess.

facilitate solving different parts of the process planning problem.

2.5. Difficult Problems in Process Planning

Many practical problems of CAPP are exemplars of more general abstract problems studied in the theory of computation [Aho74, Horo76]. There is a rich class of problems called *NP−complete* problems which are very hard to compute in the following sense. Given the size of the problem n (n may be the number of tasks to schedule, the depth of a tree to search, etc.), for all known algorithms there is no polynomial $p(n)$ that bounds the computation time needed to solve the problem. Often, the computation time grows as 2^n or worse. This means that if in t allotted seconds we could compute the solution to a problem of size n on a given computer, increasing our computer speed by a factor of 128 would only allow solution to a problem of size $n+7$ in the same t seconds! Two examples of such problems are given below.

2.5.1. The Knapsack Problem

One version of the abstract *knapsack problem* is as follows. Given a sequence of integers S = { $i_1, i_2, \ldots, i_n$ } and an integer k , determine if there is some subsequence of S that sums to exactly k. Now, suppose we buy bar stock of length k from which we have to cut pieces of length i_1 , i_2 , etc. Assume cuts are of zero length. From theory we know that searching for a subset of the n pieces to exactly use up the bar will be a hard computational problem because the knapsack problem is known to be *NP−complete*.

2.5.2. The Equal Execution Time Scheduling Problem

Another *NP−complete* problem is the *equal execution time scheduling problem* [Aho74]. Here we have a set of tasks T = { $J_1, J_2, \ldots, J_n$ }, a time limit t, a number of processors (or machines) p, and a partial ordering of the tasks in T representing the prerequisite relationships of the tasks. Assuming that each task J_i takes one unit of time and that all processors are alike, the problem is to schedule each task to some time point in 1, 2, ... , t such that the required task ordering is satisfied. If $J_i < J_j$ in the partial ordering, then J_j must be scheduled at some point after J_i. Intuitively, the scheduling of manufacturing operations to machines is even harder because all operations do not take the same amount of time, all machines do not perform the same operations, and machines may break down.

2.5.3. Search Combinatorics Problems

A persistent problem in process planning is finding the optimal product cost process plan. Often the problem arises from a large search space of possible process plans; at other times it is because of changing conditions on the shop floor. Organizing the search space and heuristics that find a near optimal solution is an ongoing task. For a concrete example similar to those already used in some systems [NaCh85, Tsat87], suppose that machining operations have been hierarchically organized into classes that contain *roughing* and *finishing* operations. By requiring that all roughing operations be done before all finishing operations, the overall scheduling problem (say order 2^n) may be reduced to two scheduling subproblems (each of order $2^{n/2} \ll 2^n$).

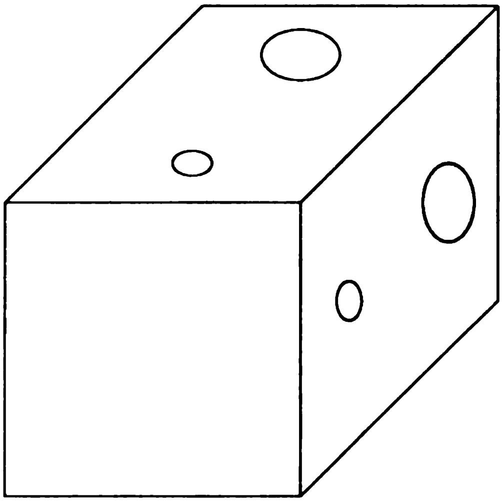


Figure 2-10: Block with four holes in it.

Figure 2-10 shows a simple product where many alternative part creation sequences are possible. Assume we are to create four holes in a rough cut block (with an initial 0.1 inch surface tolerance). A 1/4 and 1/2 inch hole are to be made on each of the two faces that are perpendicular to each other. If we require that all six sides of the block be finished to within a tolerance of 0.05 inches then a total of 10 operations are to be performed. This includes six faces to finish as well as 4 holes to be made. If any order is possible this means that we have 10 ! or 3,628,800 different ways to order these operations.

Not only are there over three million ways to order the 10 operations, but each operation can usually be done several different ways. For example, it would be possible to make the 1/2 inch holes by:

1) Drilling a pilot hole and then boring to make a 1/2 inch hole;

2) Drilling a pilot hole and then a 7/16 inch hole before reaming it to tolerance;

3) Drilling a pilot hole and then a 7/16 inch hole before boring it to tolerance;

If each macro operation can be done with three different sequences of primitive machining operations then we would have $3^{10} * 10!$ possible combinations, which is greater than 10^{10}.

The search space size for a complete search, even for such a simple object, is beyond the power of even the fastest computers. Its exponential nature makes the problem impossible to solve in the general case. This type of problem can be tackled using constraints and heuristics to prune the search space and avoid checking unnecessary, redundant, or impractical solutions.

Human process planners solve these problems using constraints and *heuristics* (also called rules of thumb). If grinding the block faces to tolerance is not necessary for accurate hole placement, then they could be done last. Using two heuristics:

1) do face finishing last if possible
2) face finishing order does not matter

yields four different operations to be ordered, or 4! possible orderings. If the holes do not intersect each other and are not close to one another then the order of drilling the holes may not matter. Usually it is best to make similar sized features (in this case holes) at the same time. If the holes can be drilled without precise fixturing and it does not matter which holes are drilled first, then the 1/4 inch holes could be made first (in any order), followed by the 1/2 inch holes. This leaves us with one good ordering of the operations after applying five heuristics. The next step involves selecting the best possible operation sequence for this ordering. Assuming three operation sequences per feature this would leave us with nine sequences to check (three for each of the 1/4, and 1/2 inch holes and three for the side finishing operation).

2.5.4. Concurrent Design Issues

Ideally, process planning and production planning would be integrated so that feedback from the current job shop load on different machines could be used in changing the cost of manufacturing parts used in the creation of the process plan. This would produce more cost effective plans on average. Allowing for flexible manufacturing and many different assembly orders in the process plan is another way to give more leeway to the production planning task and increase total shop productivity. For example, in the WoodMaster system discussed earlier, a five level order labeling of process steps was performed to delay some assembly choices until the part is made on the factory floor.

It would be extremely beneficial for long term product reliability if the process plan were created to allow for easy automatic part inspection during the manufacturing process. Proper inspection can avoid producing defective parts by monitoring relevant checkpoints for drift in feature location or size before the tolerance zone is exceeded.

Not only is feedback from the shop floor to the process planning stage worthwhile, but so is feedback from the process planning stage to the design stage. It could be time and cost effective to have the designer label parts as being 'fixed' if they had to be a certain size. Thus, if the size of the car axle were not fixed, an automatic process planner could suggest changes to the design that would make the end product less costly. Consider the design of an automobile trailer. After noting that the designer specified a 1.3 inch diameter axle and that the closest size of bar stock is 1.4 inches, a computer based process planner could suggest that the axle diameter be changed to 1.4 inches. This would save a costly lathe operation but require the wheel holes to be slightly larger and the axle to be heavier than originally planned.

Many robust solid modeling representations have been developed over the years and even more must be done to aid CAPP [Brow82, Crow84, Engl86, Fers86, Kak86, Lieb77, Loza76, Phil85, Requ80, Rich86, Tilo84]. Certain solid modeling representations are better suited than others for answering queries about adjacent surfaces, part center of gravity, or the presence of a 3-D feature. Thus it is imperative to continue to improve and expand solid modeling techniques. More parameters, details, and completeness need to be added to the models, along with procedures for keeping different representations consistent, and algorithms for computing answers to typical solid modeling questions.

Complete aùtomation of the process plan creation and code generation phases is another area for further research. For example, in automatic generation of optimized three axis numerical control programs for pocketing operations [Fers86], the user must specify the surfaces to be machined and the surfaces that should be used to guide the cutting tools. Freeing the designer from these and other monotonous tasks is a continuing concern.

2.5.5. Assembly Planning Problems

A valid assembly sequence is one in which the parts can be assembled through a sequence of translations and rotations without intersecting (passing through) other solid parts, the machine, or environment. In an unconstrained 3-D environment the problem of finding an optimal path of motion is intractable [Shar86]. Work on using heuristics or other non-optimal algorithms for solving the general path planning problem is needed. In addition, before the trajectory of a robot arm and gripper is planned, sufficient grasp points must be found on parts to insure that the part can be picked up and assembled.

Determining the stability of objects under the influence of gravity and various fixtures is a difficult problem. Palmer investigated the complexity of determining valid translational mobility and shows it to be NP-hard. Although Palmer finds that frictionless *potential* stability is in P, he shows that frictionless stability is CO-NP hard [Palm87]. This indicates that the more general 3-D solid geometry stability problem will be *intractable* as well. Using the center of mass and various supporting heuristics can aid in determining if an object is likely to be stable or not.

Assembled products are held together by fasteners or other interlocking relationships. Fasteners must be represented when determining valid assembly sequences. If the fact that different fasteners are used in an assembly sequence is ignored, an assembly sequence may be produced that requires the part to be moved back and forth between machines or require tooling to be changed each time a different fastener is assembled [Mil89a]. For example, assembling a product with several components may require that a subset R of components be applied using rivets while another subset S be applied using sheet metal screws. The assembly of parts requiring the same type of fastener one after another is more efficient in regards to changing tooling or moving to different machines than alternating between items from the R and S sets.

2.5.6. Geometric Representation and Reasoning

Representation of the input model, machines, stock, and other relevant parameters is a key issue for the CAPP problem as well as other AI type problems. The representation of surface finishes, tolerances, and how objects are joined is important to the success of a process planner. Research and development of feature-based modeling systems is needed and is progressing. Typical CAPP systems require the user to answer many questions about the part being manufactured. Deriving this necessary information for process planning from a feature-based CAD model would free the user from answering numerous questions.

Whenever two voids (holes, grooves, or other removal operations) intersect, it may be difficult to determine which void should be created first. There are many rules of thumb that can help decide whether it is better to perform one operation before another. One material removal operation may be less expensive or make the second operation much easier. Problems like these point to the usefulness of an expert system as part of a process planner.

It is difficult to decide from which side to drill a hole when it passes completely through an object. It may not even be possible to drill the entire hole from either side. In order to decide if an operation can be performed it is necessary to model not only the part being manufactured but also the machinery that is to perform the operation. By checking to see if an intersection between the part being machined and the tool or machine occurs during the operation, it is possible to determine whether or not the operation is physically possible.

Many of the current systems, such as those relying on the Initial Graphics Exchange Standard, used in process planning pay little or no attention to surface finishes and allowable part tolerances [Brdy87]. This should change as the new *Product Description Exchange Standard* (PDES) is developed. PDES can store not only the three-dimensional part features but also the current finish of various holes and surfaces of an object. The capabilities of the various machines in a factory must be stored in the knowledge base to aid in selecting the correct machines to do the various operations.

2.5.7. Summary of Chapter 2

Variant, generative, and artificial intelligence process planning systems were investigated to illustrate some of the difficult representation and processing issues involved in CAPP. Next, the most common types of three-dimensional representation schemes were surveyed and used to represent a cylindrical tube. Several knowledge representation schemes that have been successfully used for storing process planning knowledge were discussed. Algorithms and techniques from the areas of artificial intelligence, expert systems, operating systems, natural language processing, and VLSI design were seen to be directly applicable to the automation of the process planning task.

Theoretical considerations tell us that for complex parts or assemblies a) most existing process plans are not likely to be optimal, and b) it is probably not feasible to generate optimal process plans. This applies to plans generated by machines or people, and it emphasizes the need for using knowledge and heuristics. Heuristic knowledge giving relative values of certain actions and the acceptability/unacceptability of costs has been successfully used in searching for determining good moves in computer chess and is also applicable to CAPP.

Chapter 3

Assembly Planning

CAPP includes the task of assembly planning for products with more than one component. Many of the same issues and problems occur in processing materials and in assembly planning. For example, depending on the viewpoint taken, the act of painting a part can be viewed as a processing and/or an assembly step. Similar to the task of planning machining sequences, a large number of feasible sequences exist in the assembly planning domain. Since the feasible assembly sequences often cover a wide range of costs, it is important to find an efficient assembly sequence. Some considerations for automated assembly planning include accessibility of fasteners, determining grip points, path planning, and fixturing constraints on the assembly process.

The Automated Assembly Planning with Fasteners (AAPF) system was developed in 1988 at the Northrop Research and Technology Center while the author was a summer intern. It was created in the Symbolics Lisp environment and uses graphics code and ray casting interference checking routines developed by Hoffman [Hof89a] [3]. The main purpose for developing the AAPF system was to investigate some of the difficult decision and representation problems that arise in determining efficient assembly sequences when fasteners are present in an product. The AAPF system was, to the best of our knowledge, the first of its kind to deal with individual fasteners. Representing and reasoning about fasteners is critical for determining efficient assembly plans. For example, there must be enough clearance around the parts being joined in order for the fastening agent to be

[3] Additionally, Dr. Hoffman was consulted on various aspects of the AAPF system during its development.

applied during the assembly step. Additionally, different types of fastening methods may be performed at different locations in the factory, or different tooling may be required for different sizes of fasteners. Neglecting the reality of fasteners may produce an assembly sequence that is far from optimal.

3.1. Introduction

In the past, assembly was carried out manually and typically accounted for between 40% and 60% of total production time [Myru83]. Designing a product for assembly can reduce the number of parts and the assembly time. For example, the dot matrix Pro-Printer sold under the IBM name has 40% fewer parts than its predecessor and can be assembled in three minutes rather than 30 minutes. Assembly planning is of critical importance to CAPP. Automated assembly planning systems can be used to explore alternative sequences quickly and reliably to find efficient assembly sequences. Unfortunately, the number of possible assembly sequences can be very large, so heuristics and good search and evaluation techniques are needed.

Finding valid and efficient assembly sequences is very important in the manufacturing environment. For example, in the aerospace industry, during assembly of an airplane wing, it may be discovered that one of the remaining components cannot be mounted without removing part of the wing, inserting and securing the component, and then reassembling the wing. The costs of assembly are high enough without doubling part of the assembly sequence due to lack of foresight and proper planning.

3.1.1. Previous Work

Many automatic assembly planning systems ignore the fact that fasteners are used to hold individual components together. In order to create a more efficient and realistic assembly sequence, fasteners must be considered. Ignoring the fact that the objects are held together by fasteners, possibly of different types and sizes, can result in planning the assembly of components so that consecutive steps require different machines, work cells, or robot tooling. It would be more efficient to try to group assembly steps requiring similar tool types (e.g. 1/4 inch nut wrench, screw blade, welding rod, or riveting gun) together so that the object won't be moved from cell to cell or a single robot doesn't have to constantly switch tools.

Other considerations such as the orientation and location of the fasteners are also important in providing even more efficient assembly plans. Unfortunately, the search space is exponential in the number of components and fasteners in the object. Whenever N is not small, the problem becomes intractable. This suggests using heuristics or rules of thumb to keep from evaluating the large number of possible solutions to determine an efficient assembly sequence.

Table 3-1 describes previous work in the area of automated assembly planning that deals with fasteners or is otherwise relevant to AAPF.

Table 3-1: Automated Assembly Planning with Fasteners

Reference	Assembly Info.	Additional Info. Required	Individual Fasteners	Interference Detection
[Take83]	X & Z axes	connective relations	yes	human
[Brod87]	X, Y, & Z axes	no	yes	yes
[Ko87]	unavailable	mating conditions	no	no
[Seda87]	2-D	connected graph	yes	yes
[Yama87]	Z axis	no	yes	human
[Maki88]	Lego blocks	no*	no	yes
AAPF	X, Y, & Z axes	no	yes	yes

The earliest work we have found that deals with automatic assembly of objects using fasteners is that of Takeyama et al. [Take83]. Under the commonly made assumption that the reverse of a disassembly sequence is a valid assembly sequence, they were able to use connective relations between components to derive an assembly sequence. Connective relations included fit, taper, and other forms of contact that restrict the freedom of movement between components. These connective relations are supplied to the system by the user since the system does not attempt to derive the relationships.

More recently, an interactive assembly assistant was outlined by [Ko87] that determines an assembly sequence using mating conditions between individual components. Individual components are geometrically defined using a boundary representation scheme. Unfortunately, the mating conditions between the components must be specified by the user. In addition, the system does not perform its own interference checking, although it could be interfaced with a geometric modeler (not an easy task for most of the systems presently available). Instead, it relies upon the user to tell it when a particular assembly operation will cause an interference condition to arise.

Other systems analyze 3-D geometric models of products to determine valid disassembly sequences. Brodd [Brod87] applies the ideas of principal axes, envelopes, visibility, and previous experience to guide the system in disassembling the object. The system is partially implemented in Prolog and derives a CSG representation of the object from an IGES data file. Although Brodd used fasteners (bolts) in a simple block model, he does not say whether or not the system determined what the bolts held together. His system does not address the bolt size or location issue in determining efficient removal sequences for various sizes and types of fasteners.

Sedas and Talukdar [Seda87] have created a fairly powerful disassembly engine. However, the system requires more than structural information about the object and does not deal with individual fasteners. Additionally, assemblies must be symmetric, in the sense that all operations occur in a 2-D subspace. It requires both a 2-D cross sectional representation and a connected graph that represents how the various parts are connected together.

In [Maki88] a robot uses stereo vision to determine the structure of a simple toy block structure. A 3-D internal representation of the external sample is created from several stereo views and then used to determine a valid assembly sequence from known block dimensions that will replicate the example. This system is the most advanced in the sense that it need only be given a real-world example in order to derive and execute a valid assembly sequence. However, the domain is limited to that of stacking toy blocks (no fasteners are allowed) and doesn't appear to be easily extendible to products involving fasteners. For example, a vision system could not tell whether screws were right or left handed, or how long the screws were. In addition, other parts of the object not readily visible from any viewing angle cannot be modeled by the system.

Seiji Yamada et al. [Yama87] developed a system that generates disassembly sequences from 3-D CSG models and the functional part description of the fastening objects to extract the connections between parts. Their system is intended to be an interactive aid in helping humans disassemble machines in order to fix them. Removal of parts are allowed only along the Z-axis of the rotationally symmetric parts. Thus only 2-D cross sections need to be considered in testing for interference between a part being removed and the remaining parts. By testing each component of the part for removability, a procedural net is generated that contains the preconditions for removing all the parts. From this network Yamada et al. claim to be able to generate all possible disassembly sequences. Unfortunately, their system is restricted to cylindrical components and removal along the axis of symmetry for each of the parts. As with several of the other systems, the final stage of interference checking is done by human operators.

The DisAssembly Engine (DAE) system by Hoffman represents objects using a CSG representation and allows sequences of translations to be considered in disassembling possibly entangled components [Hof89a]. The DAE system does not deal with fasteners, but rather the interlocking physical constraints between components. The AAPF system uses the same ray casting techniques to check for part removal as used by the DAE system.

None of the previously surveyed systems deal with individual fasteners on a detailed level; fasteners are treated as if they all have the same geometry and tooling requirements. Nor do the systems try to remove bolts with similar sized heads (same tool requirements) before removing different sized bolts. At most workstations only one type of fastening operation is done. Even when quick change robot grippers or end effectors with multiple tools are used, there is some overhead cost associated with switching between the different types of tools. In addition, most of the systems rely on more information than is available from a solid modeling system. For example, Ko and Lee require that part mating information be supplied to their system in addition to a geometric model of the part. Similarly, the disassembly planner for redesign described by Sedas and Talukdar needs both 2-D cross section information and a connection graph consisting of nodes (representing parts) and arcs between them (representing connections) [Seda87]. This information must be specified by the user.

3.1.2. Automatic Assembly Planning with Fasteners

The Automatic Assembly Planning with Fasteners (AAPF) system creates a valid assembly sequence from the geometric description of a product. Unlike previous works, AAPF deals with the fastener representation and planning issues on an individual basis. Components or subassemblies in the product are held together by some type of fastener or interlocking relationships between the subparts. For each fastener in the object the components that are held together by that fastener are derived. Some fasteners, such as bolts, may only constrain the object to move along or rotate about some axis. However, the same bolt, when used in conjunction with a nut, can restrict all translational and rotational movement between the components when the nut is 'tightened'.

AAPF does not require any information beyond that provided by the geometric primitive representation discussed above. In addition, the system deals with fasteners on an individual basis in order to produce more efficient assembly sequences. By removing fasteners of similar type and size first, the system helps to minimize the number of tooling changes necessary.

After defining the general assembly problem, the approach taken by AAPF is presented. The object representation, assumptions, and major components of the system are outlined. Section 3.3 gives several assembly examples that illustrate the AAPF system before discussing the experimental results and AAPF run times. Section 3.5 presents conclusions and several areas for future work.

3.2. Approach

The assembly problem involves working from the geometric model of the object to be assembled. An object is composed of N components and M fasteners that hold the components together. From this geometric information an efficient assembly sequence of individual assembly operations is derived. Individual assembly operations may be high-level instructions such as '*insert screw A*' or '*join subassembly A and B along the X axis*', or they may include details such as trajectory paths and grip points. An efficient assembly sequence can be determined by keeping track of which objects constrain other objects from being removed and paying attention to fastener type, orientation, and location.

Each component of an object is represented in terms of CSG-type primitives that are combined by union (for positive primitives) or difference operators (for negative primitives). A component consists of one or more CSG primitive definitions that are considered as a single rigid object. Currently the AAPF system handles block, cone, cylinder, and sphere primitives. Each dimension for the given primitive type is specified along with the rotation and translation from the origin for a particular instance of a primitive.

Fastener primitives are used to define the various types of fasteners the system can handle. Currently, only the primitive types *bolt*, *screw*, and *nut* are allowed. This set could be expanded to include welds, gluing operations, riveting, etc. Determining bolt-nut relations and bolt-washer relations from purely geometric computations is not an easy task. However, this fastener information can easily be derived from a feature-based modeler or produced by an intelligent preprocessing program that 'looks' for certain typical bolt, nut, or screw geometries. The AAPF system expects the fastener type to be specified along with the fastener dimensions and orientation. An explanatory input file for the mouse model is given in Appendix I.

3.2.1. Assumptions

The AAPF system is capable of solving the disassembly/assembly problem for an object containing N nonintersecting components that are held together by M fasteners. The following assumptions have been made:

- Components are made up of cone, sphere, block, and cylinder CSG primitives combined using the union and difference operations. Primitives align with global axes.

- Components or fasteners are removable by a single translational movement along the X, Y, or Z axis. This implies that no rotations of components or subassemblies are required for disassembly.

- Components are rigid and do not interact in any way that would make the inverse of a valid disassembly sequence an invalid assembly sequence.

- Fasteners are between or pass through components and do not protrude from components to interfere with the removal of other components not held by that fastener.

3.2.2. Overview of the AAPF *Algorithm*

An outline of the procedure used by the AAPF system to determine assembly sequences is shown below followed by a discussion of why this procedure is in fact an *algorithm* as defined in the field of computer science.

```
1) Read in CSG data primitives and initialize data structures
putting the object to be disassembled on a working stack as one subassembly.

2) Loop until stack of (sub)assemblies is EMPTY.

    2.1) Take a subassembly off the stack.

    2.2) Determine connectivity of the objects and interference
    lists for each of the fasteners in the subassembly.

    2.3) Loop until nothing further can be removed from the subassembly.

        2.3.1) Calculate the connected components list.

        2.3.2) Intelligently remove fasteners until new loose connected
        components are formed. Update the connectivity and
        interference lists.

        2.3.3) Remove all loose components that can be removed while
        updating the interference lists. Place any removed
        subassemblies on the stack to be disassembled later.

    2.4) IF more than one connected component remains 'assembled'
        HALT "FAILURE - remaining components cannot be disentangled".

3) Take reverse of disassembly sequence as the assembly sequence.
```

In order for the AAPF procedure (defined above) to be considered an algorithm it must meet two main criteria: 1) It must consist of a finite number of steps with each step describing exactly what is to be performed. 2) The procedure must terminate in a finite[4] amount of time (i.e. no infinite loop can occur for any legal input).

Proposition 3.1: The AAPF procedure is an algorithm.

[4] An intuitive practical concern (but not part of the formal definition of an algorithm) is that in order to be useful an algorithm should terminate in a *reasonable* amount of time [Horo89].

Each one of the steps in the AAPF procedure can be implemented in a finite number of operations. If all of these operations can be decided or calculated in a finite length of time then the procedure terminates. The initialization step 1 simply reads in the geometric data for the product to be assembled and initializes the bounded size data structures for holding this information. It is also easy to see that step 3 can be executed in a finite amount of time (actually $O(N)$ time where N = number of assembly steps). The only steps which require a more detailed analysis are those contained in the loop structure of step 2. Each of the substeps in step 2 can be processed in a finite amount of time. With $|F|$ = No. fasteners and $|C|$ = No. components, step 2.1 takes at most $|F| + |C|$ time to copy the fastener and component information. Step 2.2 requires at most $O(|F| * |C| + |C|^2)$ time to determine which components each fastener holds and to determine the component interference lists. The complexity of step 2.3 is similar to step 2.2 since the connectivity and interference lists are updated. AAPF only allows removal sequences to consist of a single translational motion along the axis so difficulties associated with many possible removal directions, multiple translations, and rotations of components and fasteners do not arise. More sophisticated systems could face an exponential time complexity unless good heuristics are applied to solve these problems which AAPF avoids with its limited number of axial removal directions and single translational motion assumptions. Step 2.4 requires only a simple check of the number of remaining components and fasteners.

Since all the steps have been argued to take a polynomial amount of time, it only remains to make sure that all loops in the procedure will terminate in a polynomial number of iterations. In step 2, first consider the innermost loop (step 2.3). A single subassembly is processed in step 2.3. Each time through the loop in 2.3 at least one fastener (and possibly several components) will be removed from the subassembly. If during step 2.3 nothing further can be removed (and more than one fastener and/or component remains, then step 2.4 causes the program to HALT and print a message indicating that AAPF cannot disassemble the object (even though a disassembly sequence may be possible using multiple translational or rotational movements -- which is disallowed by the assumptions used in AAPF). Thus, step 2.3 will be executed a finite number of times for a single subassembly since during each execution at least one fastener and perhaps a few components may be removed, or the program halts having failed to disassemble the object. The outermost loop is performed until all subassemblies have been disassembled. The entire assembly is first placed on the stack; however, as individual fasteners are removed in the inner loop (step 2.3), it is possible that smaller subassemblies may be placed back on the stack. However, in order for a subassembly to be placed on the stack at least one fastener must be removed from the object. Since the number of fasteners and components in an object is finite, only a finite number of subassemblies may be placed on the stack for processing. It is obvious that this outer loop will only be executed a finite number of times ($\leq |F|+|C|$), ending either in a successful disassembly or halting with a failure condition in step 2.4. The AAPF algorithm will always terminate either with a valid assembly sequence or with a failure message indicating that AAPF could not find a valid assembly sequence: thus, Proposition 3.1 holds true.

An assembly sequence found by AAPF will be valid under the assumptions made by the AAPF system. i.e. AAPF will not find an assembly sequence that cannot be performed (under the assumptions used by AAPF). Additionally, if AAPF halts without being able to disassemble the object, then it will be impossible to assemble that object under AAPF's assumptions. Of course, this only holds true to the limits of the machine accuracy subject to round off error (i.e. round off error could cause the AAPF Lisp code to find an assembly solution when that solution was in fact impossible and vice versa). Of course the assumptions made by the AAPF system are in fact quite limiting. AAPF may not be able to find any possible assembly sequence for an object that could be assembled using multiple translational motions or a non-axis assembly direction. Additionally, AAPF may find an assembly sequence that would be impossible to perform under the influence of gravity (which it does not model).

3.2.3. Disassembly in AAPF

Disassembly of an element from the subassembly stack begins by determining the connectivity of its components and fasteners, and determining interference lists for its fasteners. An interference list indicates those fasteners which are accessible to be removed from the subassembly. Fasteners are removed until loose subelements are formed. Connectivity and interference lists are updated as loose subelements are removed. This process repeats until a single component remains. The following subsections discuss the interference check procedures in detail.

3.2.3.1. Part Envelope Interference Test

Determining whether or not a component or subassembly can be removed is done at one of two different levels. In order to perform quick interference checks, the system first tests for interference by comparing the primitive part envelopes for all pairs of primitives being removed and those remaining stationary, where part envelopes are defined as the largest and smallest extent of the primitive along the x, y, and z axes. If no intersection of these part envelopes occurs along a direction of the x, y, or z axis, the part may be removed along that direction. These part envelopes are calculated in the initialization stage of the algorithm for each of the geometric parts so that the information is readily available.

3.2.3.2. Ray Casting Interference Test

An envelope check may not be sufficient to determine whether or not a component or subassembly can be removed from the object. In these cases a ray casting technique is used to determine how far the one subassembly can be removed from the other before interference occurs [Hof89a]. A two-dimensional grid of points is chosen that covers the component(s) being removed. For each of the points on the grid a ray is cast along the axis of removal. Intersections between a ray and any of the component(s) not being removed is noted. The shortest distance any of the rays travel before hitting an object is the maximum distance the component(s) may travel before hitting another component. AAPF considers fasteners to be removable if they have sufficient clearance above the top of the fastener and if a box test for robot arm accessibility is passed (as discussed below). A component is considered removable only if all of the rays along one of the axis directions do not hit any objects. This simple component removal strategy was used in AAPF since its purpose is to reason about the removal of fasteners.

An important parameter in the ray casting technique is the grid spacing. If the grid spacing is too large, it is possible that certain object features will not be detected as causing interference between the components. Additionally, a small grid spacing results in time consuming computations that are not necessary. AAPF currently has a fixed grid spacing which is assumed to be small enough to detect all object interferences. It is possible to determine the grid spacing automatically based on the size and location of components in the object as discussed in [Hof89b].

If currently unremovable fasteners still remain in the assembly, and the system cannot remove any more components in either of the above ways, it tests to see if any components or subassemblies can be slid off of bolts without nuts on them. Items on a bolt without a nut constraining them may be slid off in the one axial direction away from the bolt's head. This sliding test is done only when the system would otherwise report failure since it is more natural and efficient to remove bolts instead of sliding the objects or subassemblies off the bolts.

3.2.3.3. Fastener Removal

The AAPF system needs the bolt, screw, and nut functional designations to automatically derive the connectivity between components in the product being disassembled and find which object components interfere with fastener removals. Ray casting is done in the direction of the fastener's axis of symmetry at four points to determine how much clearance above the fastener exists, as well as which objects the fastener passes through (or holds). The four points are chosen to be centered around the fastener's removal feature (e.g. bolt, nut, or screw head) and slightly larger than the fastener diameter to account for the size of the removal tool. A fastener holds all the objects that it passes through or constrains. Bolts only constrain objects to motion along their axis of symmetry unless a nut is on the bolt restraining the objects between the bolt head and nut. Screws restrain all the objects they pass through.

The presence of one or more nuts on a bolt is detected automatically by the AAPF system from the physical locations of the fasteners and is not specified in the input file. A bolt cannot be removed until any nuts on it are removed. The AAPF system uses ray casting interference checking routines and a quick robot arm access check to determine which fasteners can be removed.

Fasteners are checked for freedom of removal along their axis of symmetry away from the component(s) they are in or are holding. Each fastener is considered a candidate for removal if enough free space exists above the fastener for the removal of its shank (bolt or screw) and enough free space around the fastener's head exists. If enough room exists for the fastener to be withdrawn from the objects it is holding, an additional check is made to see if this location can be reached by the robot arm. Currently only a simple check is made to see if a small cube can be removed from the position of the removed bolt along one of the six primary axis directions. A more complex testing routine could be used once a specific robot manipulator and arm geometry were known. Specific removal tests are as follows:

Screws can be removed if the clearance (empty space) directly over the screw head is greater than the sum of the length of the screw and a constant clearance value. The constant clearance value accounts for the height of the screw removal tool and robot arm during the removal process.

Bolt removal is defined similarly with the added restriction that no nuts remain on the bolt.

Nuts can be removed if the amount of open space above the nut (after the end of the bolt) is larger than the sum of the nut height and a constant value that accounts for the nut removal tool. Currently, access to the bolt head that the nut is on is not checked in AAPF. It is assumed that the bolt will not turn when the nut is being removed.

To improve the efficiency of the assembly sequence, the system keeps track of the type, size, orientation, and location of the last fastener removed. Every fastener that can be removed is evaluated based on the characteristics (type, size, orientation, and location) of the fastener that was previously removed. Fasteners of the same type are removed first. If fasteners are of the same type, size is the next criterion used. If several fasteners are of the same type and size, then the orientation of each of the fasteners is used. When all other factors are the same, the locations of the fasteners are compared to the location of the last fastener removed. This selection algorithm helps to minimize the number of tooling changes required as well as the distance a robot arm has to travel. Although the sequence is not guaranteed to be optimal, on average it will be much more efficient than a random choice of the next fastener to be removed.

3.3. Experimental Results

The Automatic Assembly Planning with Fasteners system is written in Symbolics Common Lisp for a Symbolics 3600 series workstation (running under Genera 7.1). The AAPF system consists of 4500 lines of Common Lisp code with an additional 1000 lines of code describing the assemblies used in the development and testing of AAPF. The system has been tested for object geometries ranging from simple overlapping board structures to more complex objects such as a candelabra, desk, and even a mechanical mouse. The AAPF system shows the current status of the disassembly process in a graphics window. As it considers components or subassemblies for removal from the remaining components, it displays the components to be removed using dashed lines. This slows down the search for a disassembly sequence, but is useful for showing the user how it arrives at the disassembly sequence. The next section contains Table 3-2 that summarizes component data and results for four of the models given to the AAPF system. The run times given are the average of three identical runs. Depending on the influence of garbage collection, object viewing angle [5], and other multitasking environment variables, the run times can vary considerably.

3.3.1. Models Used

The four examples discussed in this thesis vary in the quantity and type of primitives and fasteners that comprise the object geometry, as well as in the difficulty of obtaining disassembly sequences. Figures 3-1 (a-d) and 3-2 (a-b) are wire frame renditions of some of the object models used with the AAPF system.

Table 3-2: Object Table and Run Times [6]

Object	No. of	Primitives			Ray Casting	Run
Name	Parts	Components	Fasteners	Total	Checks	times
Blocks	5	6	8	14	0	35 sec.
Robot	8	10	6	16	0	35 sec.
Candelabra	14	22	8	30	0	1.5 min.
Desk	12	26	24	50	0	3 min.
Mouse	14	42	5	47	5	6 min.

The blocks model (Figure 3-1 a) and robot model (Figure 3-1 b) both have a small

[5] A change in the viewing angle may require more computations and display updating time.

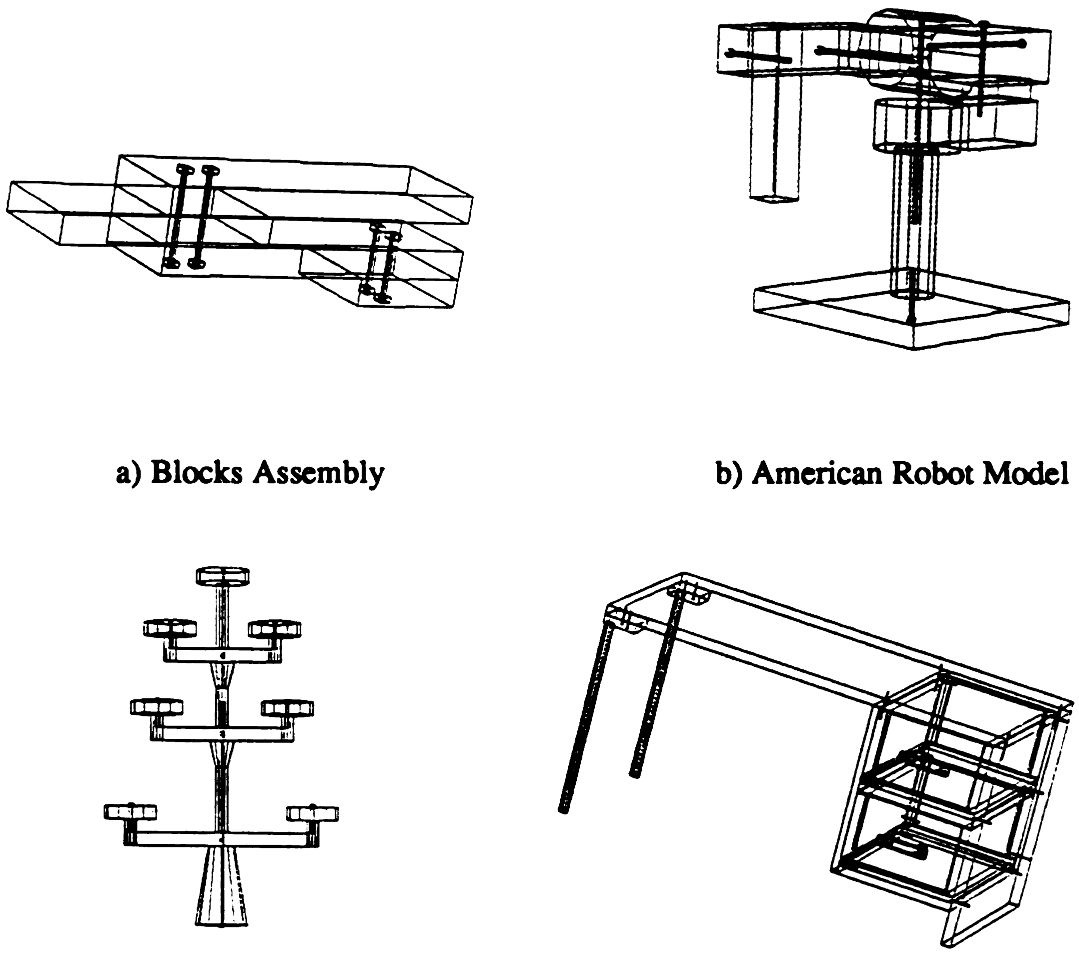
[6] AAPF was written in Lisp for a Symbolics 3600 workstation. The run times include the computational overhead associated with displaying the disassembly process in a graphics window.

number of components and fasteners. The simple geometry of these models allowed AAPF to find assembly sequences relatively quickly. The candelabra (Figure 3-1 c) has a larger number of components but is still of simple construction. Unfortunately, the AAPF system does not know about the stability of objects. Thus, when it disassembles the candelabra, the large bottom center screw is removed first, creating several loose components and subassemblies that would tumble downward due to the forces of gravity. This is not an easy problem to solve [Palm87].

Although the desk (Figure 3-1 d) has the largest number of fasteners, its components are of simple block and cylinder type. This allows AAPF to compute the removal and interference checks using the enclosing part envelopes instead of using the more time consuming and exact ray casting routines. By choosing the next fastener to be removed based on the type, size, orientation, and location of the fastener with respect to the last fastener removed, the system chooses a disassembly sequence that helps minimize tooling changes and the distance traveled between fasteners to be removed.

Of all the objects assembled, the mouse (Figure 3-2) was the most complex, using all the primitive types: sphere, cylinder, cone, and block. Its closely interlocking structure caused the enclosing envelope tests to fail on numerous occasions and the more time consuming ray casting interference check to be applied. In testing to see whether the large outer plastic case of the mouse could be removed, the ray casting routine took slightly under 3 minutes of cpu time. This accounts for nearly 50% of the time it took the AAPF system to disassemble the mouse. Not only does the large area of the mouse case give rise to using a relatively large number of rays, but each of the rays must be checked for intersection with each of the other component primitives that are not being removed with the plastic case. A disassembly sequence produced for the mouse is shown in Appendix I.

Changing the order of the fasteners in the input file can change the evaluation and removal sequence for an object. This is due to the fact that often many equivalent



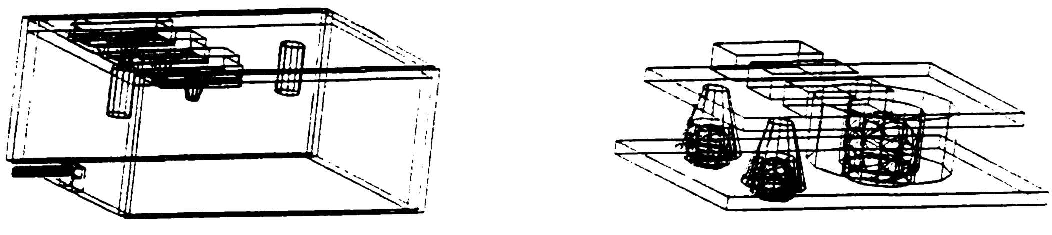
a) Blocks Assembly

b) American Robot Model

c) Candelabra

d) Student's Desk

Figure 3-1: Object Models Assembled by AAPF



a) Outer Mouse Parts

b) Inner Mouse Parts

Figure 3-2: The Mouse Assembly

assemblies (disassemblies) exist, and depending on which part of the search tree is entered, the AAPF system may take more or less time for a given assembly solution. When the ordering of the two screws that hold the black plastic mouse case to its base is reversed, the time it takes the system to determine an assembly sequence is 9 minutes, a 50% increase in run time.

3.4. Evaluation of Process Plans

Many of the same issues are involved in determining efficient assembly and machining/processing sequences. In fact, since both machining and assembly steps may be intermixed on the shop floor, a single set of evaluation functions is appropriate for evaluating both the processing and assembly sequences which make up the final process plan. This section discusses evaluation functions for rating the final process plan which may consist of both machining and assembly steps. These evaluation functions are used to rate hand generated assembly plans and assembly plans generated by AAPF as well as other hand generated process plans. The evaluation results are given in Table 3-4 to illustrate the wide range of time requirements for different process plans.

A product can usually be created and assembled in many different ways, with some solutions being much better than others. AAPF assumes that the parts have already been created and only need to be assembled. Additionally, AAPF assumes that it produced an efficient solution the first time it was executed, but AAPF could have produced several possible solutions to be evaluated further. This section investigates several possible process plans for different products and rates each plan based on the time required to create the final product. The manufacturing time is highly dependent on the factory environment in which the product is created.

The cost of a process plan depends on the machining and assembly methods used. The economics of manufacturing have been investigated in several books [Boot82, Nevi89]. Boothroyd concentrates on the assembly planning problem by examining the benefits and costs associated with a human operator assembly line, a dedicated hybrid machine, a free-transfer machine with programmable workheads, an assembly center with two robotic arms, and a *Universal Assembly Center* [Boot82]. Different restrictions and assumptions on allowable assembly steps occur depending on which type of assembly method is used. Nevins and Whitney discuss both manufacturing and assembly issues [Nevi89].

Different process plans can have significantly different costs. Appendix II contains two different assembly sequences for a board and fastener assembly (o1, o2) and three different assembly sequences for the mechanical mouse models (m1, m2, m3) used by AAPF. Additionally, several process plans for a block with holes and the toy car model are shown. The assembly sequences o1 and m3 were generated by AAPF (with hand made additions to account for refixturing requirements), and the remaining assembly plans were generated by hand. AAPF produced the "best" plan for the board and fastener assembly; however, the assembly plan for the mechanical mouse required 54% more time than a plan which was generated by hand. The longer time for the assembly sequence generated by AAPF is due to the fact that AAPF did not consider refixturing steps in generating its assembly plan. However, to show the importance of fixturing in assembly planning, the process plans (given in Appendix II) that were evaluated to obtain the results in Table 3-4 take into account refixturing steps. The mechanical mouse can be assembled in over 54 million different ways (see Section 4.6.4), so finding an efficient solution may be very time consuming if a large number of plans must be evaluated.

3.4.1. Manufacturing Operation Time Costs

In order to evaluate the different processing and assembly sequences a set of assumptions and time cost functions for the manufacturing operations must be made. The manufacturing environment used to evaluate the assembly plans is shown in Figure 3-3. The manufacturing environment includes a single three axis robot arm that can have mul-

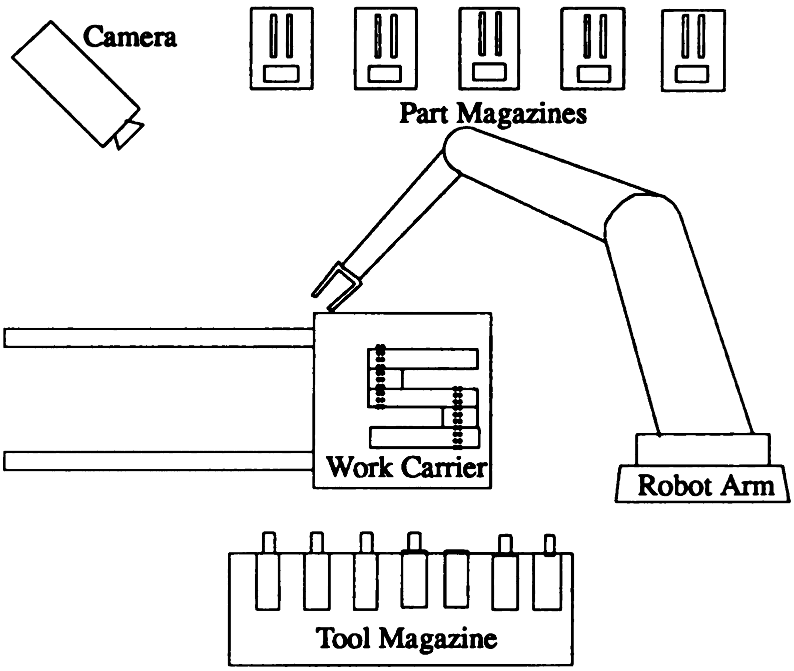


Figure 3-3: Manufacturing Environment Model

tiple end effectors (tools). The part gripper, nut fastener, saw, drills, and screw gun end effectors are available from a tool magazine along the edge of the work cell. Large parts and subassemblies are brought into the center of the 3'x3' work environment on a work carrier which has a *Universal* part fixture on it to hold the product as it is being manufactured and assembled[7]. Programmable part magazines for the various fasteners and small subparts of the product are located along the perimeter of the work cell.

[7] Either this *Universal* part fixture is adjustable and capable of holding any of the possible subassemblies, or a set of fixtures is on the work carrier that can hold any of the subassemblies encountered during the assembly process.

Table 3-3: Manufacturing Operation Time Costs

Work Cell Operation	Cost in Distance and time
Re-fixturing assembly	5 seconds
Tool Change	3 feet + 1.5 second
Part Acquisition	3 feet + 1 second
Insert Screw	2 seconds
Drilling or Sawing	2 seconds
Assemble Part	3 feet + 2 seconds
Fetch & Fasten	3 feet + 2 seconds
Drill Hole	2 seconds
Make Cut	2 seconds

Table 3-3 gives the costs for each of the eight allowable manufacturing operations. The manufacturing costs are given in terms of distance traveled by the robot arm and any additional fixed amount of time required for the operation. For example, the time for a tool change (3 feet + 1.5 second) assumes that the robot starts in the center of the work area (3'x3' table) and moves to the perimeter to switch tools before moving back to the center of the work area for the next step. Plans rated highly using these approximate cost functions could be reevaluated using more detailed evaluation functions that plan the actual distances moved and the time required for each step. The approximate cost functions can be used to evaluate the different time costs associated with fast and slow robot arms. For example, the tool change operation takes 3 + 1.5 = 4.5 seconds for a 1.0 foot/sec robot arm or 1.5 + 1.5 = 3.0 seconds for a 2.0 feet/sec robot arm.

3.4.2. Discussion of Evaluated Process Plans

Each of the nine process plans were evaluated according to the evaluation functions described in the previous section. Table 3-4 shows the estimated cost in seconds for the different mechanical mouse and board assembly plans, as well as the block and toy car process plans. Each plan was evaluated twice, once with robot arms that move 1.0 foot /

second and then again with a robot arm speed of 2.0 feet / second. It is interesting to note that the heuristically generated assembly solution generated by AAPF is in fact the best solution for the overhanging board assembly. The second overhanging board assembly sequence o1 illustrates a bad assembly sequence that requires 65% or 76% more time than the AAPF solution depending on the robot arm speed.

Table 3-4: Comparison of Process Plan Manufacturing Times

Plan Name Number	Created by	Time Cost 1 ft/sec	% Longer than *Best** Plan	Time Cost 2 ft/sec
Overhang o1	AAPF	87.5	0%	63.5
Overhang o2	hand	144.5	65%	111.5
Mouse m1*	hand	116.5	0%	86.5
Mouse m2	hand	159.5	37%	123.5
Mouse m3	AAPF	169.5	45%	133.5
Toy Car c1*	hand	231.5	0%	176
Toy Car c2	hand	465.5	101%	348.5
Block b1*	hand	53.5	0%	43
Block b2	hand	109.5	104%	87

* "Best" plan done by hand; not by exhaustive search.

It is important to note that the evaluation functions and evaluated plans used here take into account part fixturing and assembly environment factors not modeled by the AAPF system. This accounts for the poor rating of the AAPF assembly plan m3, which has the longest execution time of the three example mechanical mouse assembly sequences investigated. AAPF did not consider fixturing constraints since it assumed that the product was suspended in the middle of the work environment. When fixturing is considered, the AAPF sequence of assembly steps requires a large number of refixturing steps resulting in 53% more time to assemble the product than the efficient assembly sequence m1. These assembly sequences show the importance of modeling all aspects of the work cell environment. The reality of fixturing, tooling, stability of subassemblies, path planning, and parts accessibility must be considered in optimizing real-world assembly sequences.

The four process plans for the toy car and block with six holes were all created by hand. They illustrate that time differences of over 100% are possible between good and bad process plans. This emphasizes the importance of finding efficient process plans from the set of all possible plans rather than accepting the first feasible plan found. This large difference in time costs can be attributed in part to the larger number of tools (as compared to the first five assembly plans) used in the process plans which allow for inefficient plans to incorporate a large number of tool change operations. Finding an efficient sequence can be very difficult without appropriate heuristics and search techniques when such a large number of solutions exist.

Although AAPF tried to minimize tool changes, it did not account for fixturing requirements. Some of the plans generated by AAPF are optimal (such as o1), while others require different fixturing (m3) which causes the AAPF plan to be inferior to the m1 and m2 assembly plans. Considering individual fasteners is important to insure that any derived assembly sequence can be created, minimize tool changes, and fully investigate fixturing concerns.

3.5. Summary of Chapter 3

The Automatic Assembly Planning with Fasteners (AAPF) system was developed to investigate some of the difficult representation and planning problems occurring in CAPP. AAPF is able to determine a valid disassembly sequence using only the geometric descriptions of the components and labels for the different fastener types. The reverse of the disassembly sequence is a valid assembly sequence under the assumption that no external or internal forces affect the object components and subassemblies. For each fastener it determines which components are restrained by that fastener. Using this information, the system maintains the connectivity of the various components during the disassembly process. By paying attention to the fastener type, size, orientation, and location, the system produces efficient disassembly sequences by minimizing the number of tool (or workstation) changes and by using heuristics to reduce the distance traveled between successive fastening operations.

AAPF can determine a valid assembly sequence (if one exists) for objects with screw, nut, or bolt fasteners that can be defined in terms of Constructive Solid Geometry primitives. The disassembly plan can be derived in a reasonable amount of time for even complex objects such as the mouse. The mouse assembly sequence is derived in 6 minutes. This is fast enough so that a designer could watch the process, and perhaps notice that by changing the design slightly a better assembly sequence would be possible. Assembly planning times of several hours would still be useful (compared to weeks), since after planning the assembly sequence, it could be graphically displayed in order to determine if design changes would speed up the manufacturing process.

Although we assume that primitives and components are aligned with global axes, AAPF will be able to handle cases where this is not the case. However, the quick enclosing envelope removal check used in AAPF will be overly conservative and may indicate interference between an object being removed and another stationary object when no interference would really occur. This may cause the more time consuming removal check to be performed. In addition, AAPF assumes that installed fasteners do not protrude so that interference checks between components being removed and fasteners not being removed does not have to be performed. This non-protrusion assumption could be changed, at the expense of speed, by requiring interference checks with installed fasteners that are not being removed.

Machining and assembly plans are evaluated for different products in a flexible manufacturing environment. Differences of up to 100% are found in the time necessary to manufacture and assemble a product, depending on how efficient the derived process plans are. The evaluation functions take into account fixturing constraints so it is not surprising that the AAPF plans are not always efficient, since they did not consider fixturing constraints. The reality of the manufacturing process shows the importance of fixturing constraints for any machining and/or assembly planner. It is also vital to reason about fasteners when fixturing constraints are considered. For example, although a component may be able to be placed into an assembly, the fixtures may interfere with the process of fastening the component to the subassembly, making such an assembly step impossible.

In order to be able to define a wider range of manufacturing products, a larger selection of fastening primitives is necessary. For example, welds, rivets, nails, glue, and other commonly used methods of fastening objects should be represented. Although fasteners such as welds and rivets are not thought of as being easy to disassemble, their computer representations can be treated as such. This allows us to be able to continue to use the reverse of a disassembly sequence as a valid assembly sequence.

In the long term, work needs to be done in dealing with internal and external forces on the subassemblies and accounting for part fixturing. In several of the examples the disassembly sequence would have caused the remaining components to be unstable, perhaps causing components to fall. Unless multiple robot arms are used or special fixturing is designed, the current systems (including AAPF) that generate assembly sequences will fail when working with larger, less restricted product domains. The only automated assembly system to my knowledge that has addressed this issue (although in a limited domain) is the ARI robot for assembling toy blocks [Maki88]. This problem is very difficult for arbitrary product shape and composition [Palm87] and promises to remain a research topic for the near future.

Chapter 4

Representing and Sequencing Constrained Tasks

The number of possible task orderings can range from 0 to N! for a set of N tasks with constraints among the tasks. Once the number of possible task orderings is known, appropriate heuristics and/or search and evaluation techniques can be applied to help insure that an efficient process plan can be found in a reasonable amount of time. Precedence graphs are used to represent the tasks and constraints. An algorithm is presented that acts as an *oracle* by determining the number of sequential orders possible in a precedence graph.

A subgraph replacement strategy is applied that determines the exact number of task orderings for a class of *N-fold* graphs or upper and lower bounds on the number of orders for non-*N-fold* graphs. The Oracle algorithm always completes in low order polynomial time even when there are an exponential number of orders. The algorithm is applicable to many practical problems in CAPP and should be of value in searching for efficient sequences of operations.

4.1. Motivation

Often in Manufacturing Planning or Artificial Intelligence domains a set of tasks is to be performed without violating a set of precedence constraints among the tasks. The goal is to find an optimal or efficient task ordering. Unfortunately, there may be a large number of possible task orderings for a fixed set of tasks and constraints. The number of task orderings for N different tasks can range from 0 orders if the constraints do not allow a single task ordering, to N! orders for a completely unconstrained set of tasks.

There may be many sequential orderings of the tasks that satisfy all the precedence constraints, each ordering having an associated cost or merit, making it difficult to determine an optimal or at least an efficient ordering of the tasks. For example, the door lock assembly shown below in Figure 4-10 has 14 parts and 15 precedence constraints that allow for 308,880 different assembly sequences. It is interesting to note that the number of possible orderings for 14 tasks can range from 0 to 14! = 87,178,291,200[8]. In the door lock example, it may not be feasible to evaluate every possible sequence in detail to find an optimal solution.

DeFazio and Whitney have found that significant cost differences can exist between alternate manual assembly sequences of automobile subassemblies. Depending on the subassembly under consideration, they found different assembly orders with cost differences ranging from 5% to 20% [Whit88]. Determining an optimal or at least efficient assembly ordering when many possible sequences exist can keep manufacturing operations competitive and add to the profit margin.

One way to find an optimal sequence for a set of tasks is to enumerate and evaluate all the possible task orderings. Unfortunately, evaluating each possibility in detail is impractical when a large number of task orderings exist. Heuristics must be accurate (admissible) in the evaluation of partial sequences to avoid discarding the best solution(s). If an admissible heuristic is available, it may be possible to look at only the most promising task sequences and still find the best solution. However, the heuristic must prune enough of the search tree to keep from evaluating a large number of solutions. In general, the search space is exponential in N and hence complete enumeration is intractable.

During the development of the AAPF system, it became apparent that knowledge of how many different plans were possible for a set of disassembly tasks would be useful in

[8] Zero valid orders corresponds to having a directed cycle in the precedence constraints.

searching for efficient plans or in choosing a flexible set of plans. For example, when a single component of an assembly is considered for disassembly in a given direction, several other components may block its path. This creates precedence constraints between the removal of the single component and each of the other components along its removal path. Given a primary removal direction, geometric reasoning can determine the precedence constraints between the various components in an assembly. This derived precedence information can be used by the Oracle algorithm (as discussed in this chapter) to determine the number of possible assembly task orderings. The number of possible assembly task orderings can be used to choose an appropriate search and evaluation technique for finding efficient assembly plans. Alternatively, the number of possible assembly orderings can be used to compare different assembly directions in order to choose a direction with the largest number of possible assembly orderings (indicating more flexibility for the assembly of the components [Home90]). The usefulness of the Oracle algorithm is not limited to process planing endeavors but can be applied to general task planning problems as well.

Suppose we had a program that could act as an oracle and tell us how many different valid task orders were allowed by a set of precedence constraints. Based on the number of different orderings, we could decide to evaluate all or only a portion of all possible task orderings. For example, if we know that 8 different orderings for a set of tasks are possible, we could evaluate them in depth and choose an *optimal* solution. However, if over 3 million valid orderings exist, then an appropriate heuristic, A^*, or branch and bound search, etc. could be applied to find an efficient ordering for the tasks. Depending on the number of possible orderings, a range of heuristics or different search and evaluation techniques could be applied.

In assembly planning it may be desirable to have a fixed set of assembly tasks which are all executed, but in varying order. This allows for flexibility in adapting to the random arrival of parts or to accommodate other changes in the assembly environment.

Homem de Mello and Sanderson have worked on this issue for AND/OR graphs. Depending on fixturing or other constraints, many different sets of assembly tasks (assembly plans) may be possible and the assembly plan that allows the maximum number of different sequences (maximum flexibility) is preferred [Home90].

If the system can quickly determine how many task orderings are possible, then given the geometrical, stability, and other constraints involved in the problem it can report to the user how many task orderings are possible. With this information the user (or the system) could selectively add constraints to reduce the number of possibilities to be considered. Alternatively, constraints could be deleted to create a more flexible set of assembly possibilities. The Oracle algorithm could be helpful in such an application with a fixed set of tasks.

Other researchers have developed tools for the enumeration and evaluation of alternate assembly sequences [Home89,Whit88]. DeFazio and Whitney enumerate the possible assembly orders using *diamond* diagrams to represent the assembly states. Each assembly state denotes which liaisons have been connected at that time in the assembly. Unfortunately, the number of states in an N part diamond diagram can be up to the total number of partitions possible for a set of n elements, $\Sigma_{i=1}^{N} \left\{ {N \atop i} \right\}$, with the number of edges being as high as $\Sigma_{i=1}^{N} \binom{i}{2} \left\{ {N \atop i} \right\}$ [Wolt90]. Obviously, counting the number of possible orders using diamond diagrams is intractable in the general case.

One goal of this research is to produce a software package that can serve as an oracle in determining the number of possible task orderings given a set of tasks and precedence constraints *without* enumerating all the possibilities. In the general case this problem is computationally intractable. However, working from precedence graph representations of the tasks and constraints, the *Oracle* algorithm can produce exact answers in low order polynomial time for *N-fold* graphs. Non-N-fold graphs are also evaluated in low polynomial-order time, however, bounds on the number of orders are returned instead of an exact value.

In general, precedence constraints may contain AND (Λ), OR (V) and $\leq$ relationships. The basic Oracle algorithm works with sets of simple precedence constraints of the form: task A must precede task B ($A < B$). Section 4.3.1 discusses how a set of tasks and general precedence constraints (containing Λ, V, and $\leq$) are broken down into one or more sets of simple precedence constraints on the set of tasks. Once the tasks and precedence relationships have been translated into sets of simple precedence constraints the Oracle algorithm can be applied to the individual sets of simple precedence constraints and the results merged to get the number of possible task orderings.

4.2. Background

To avoid time and space problems that occur in enumerating and storing loosely constrained assembly plans, the Oracle algorithm uses a constraint-based representation. The Oracle algorithm requires a set of tasks and precedence constraints as input. Today most of the assembly tasks and precedence constraints are determined by an expert human process planner, but the field of Computer Aided Process Planning is attempting to automate this process. For example, in assembly planning, methods for determining the precedence constraints have been investigated by a number of researchers [Bour84, DeFa87, Huan89].

At least one valid ordering can be found (if it exists) by applying a topological sort (topo-sort) algorithm on a given set of constrained tasks. A topo-sort can be performed in O(|Tasks|+|Constraints|) time [Horo76]. Although the algorithm can efficiently find a single valid order (if one exists), it does not determine how many different orderings are possible. To obtain the number of possible orders, the topo-sort routine could be modified to generate each valid order only once as the routine was called successively. Then this routine could be executed repeatedly to find each possible ordering. (These orderings could simply be counted, or could even be evaluated for cost.) While some researchers have discussed generating all the possible assembly sequences [Home89, Seki88], enumerating all the assembly sequences (up to N ! sequences for N tasks) can be too time consuming for a large number of tasks.

Homem de Mello uses several heuristics for evaluating and selecting among different assembly plans in an AND/OR graph [Home90]. His W_1 heuristic determines the number of possible assembly sequences for the tree representation of an assembly plan. The plan with the largest number of possible orderings can be selected in an attempt to find the most flexible plan. The parallel reduction formula in Section 4.4 is similar to the W_1 evaluation function, except that we apply the formula to non-tree structures[9].

Although enumerative data structures have many uses, they can be time consuming to generate and may also require large data structures to store all the possible assembly orders. The worst-case data structure size requirements for six enumerative data structures has been investigated by Wolter [Wolt90]. With 15 parts in an assembly, the worst-case space requirements range from 10^7 for AND/OR graphs to 10^{20} for storing all possible state sequences.

Determining how many different orderings are possible given a set of tasks and precedence constraints is known in mathematics as finding the number of linear extensions of a poset [Stan86, Riva84, Wink90]. Winkler has shown that in the general case determining the number of linear extensions of a poset is #P-complete [Wink90].

Definition: #P (sharp P)

#P is a class of counting problems in which a single instance can be computed by a nondeterministic polynomial time Turing machine. However, instead of deciding whether or not a solution exists, #P problems have the additional task of returning the

[9] If an *END* vertex (Section 4.5.2) is added to an assembly plan (tree) representation it becomes an N-fold graph.

number of solutions.

Definition: #P-complete (sharp P-complete)

A problem is #P-complete if it is in the #P class and can be reduced to one of the *toughest* problems in the #P class, i.e. be transformed into a known #P-complete problem such as counting how many different satisfying truth assignments exist for an instance of the SATISFIABILITY problem [Gare79].

No polynomial time algorithm is known for solving any of the problems in the #P-complete class. Since determining the number of possible orders is *intractable* today, (and will likely remain an *intractable* problem), the approach taken is to find a polynomial time algorithm that can perform as an oracle on a large number of practical cases. Some classes of graphs can be solved in polynomial time. For example, an algorithm for partial orders of a bounded width k has a time complexity of $O(n^{k*k})$ and space complexity of $O(n^{k*k-k})$ [Atki86]. Another algorithm for posets with bounded decomposition diameter d has a time complexity of $O(n^{d*d})$ [Habi87].

The first version of the Oracle algorithm was capable of exactly determining the number of possible task orderings for a class of *simple-fold*[10] graphs [Mil90a]. The latest version uses the concept of subgraph replacement to expand the set of graphs that can be handled exactly to the class of *N-fold* graphs as defined in Section 4.3.3 [Mil90b]. The domain of graphs that can be handled exactly by the new Oracle algorithm is now larger, and it produces both lower and upper bounds on the number of possible orders instead of a single estimate as the first Oracle algorithm did. The new Oracle algorithm returns either an exact answer for *N-fold* graphs or upper and lower bounds for non-N-fold graphs that cannot be reduced exactly.

[10] The class of simple-fold graphs is also known as series-parallel graphs in mathematics.

4.3.

po

bli

sp

as

g

c

v

4

s

g

t

i

le

nu

gle

set

ship

num

time

tasks

tionsh

4.3. Representation

Enumerative representation schemes suffer from the potential problem that all the possible assembly states must be enumerated and stored. For loosely constrained assemblies with many assembly tasks, any enumeration scheme will have prohibitive time and space requirements. The approach taken is to work with the precedence constraints in an assembly problem. The Oracle algorithm works on precedence graphs consisting of the generic tasks (which may be manufacturing or assembly tasks) to be performed and the constraints among these tasks. The assembly tasks to be performed are represented as vertices in a directed graph(s).

4.3.1. AND (Λ), OR (V), and $\leq$ Relationships

In general, precedence constraints may contain AND (Λ), OR (V), and $\leq$ relationships requiring more than one precedence graph for representation. Converting from general precedence constraints into sets of simple precedence constraints is discussed in this section. Λ, V, and $\leq$ relationships in a set of precedence constraints can be converted into several subproblems containing only simple precedence constraints. These subproblems are evaluated independently with the results being recombined to find the total number of orderings possible. Each set of simple precedence constraints represents a single precedence graph. Unfortunately, the number of precedence graphs represented by a set of general precedence constraints is exponential in the number of Λ, V, and $\leq$ relationships. This indicates that the Oracle algorithm is best suited for problems with a small number of Λ, V, and $\leq$ precedence relationships and cannot return advice in polynomial time for the most general problem.

Mapping from Λ, V, and $\leq$ relationships to multiple precedence graphs for pairs of tasks is outlined below. Extensions to sets of three or more tasks and different interrelationships can be arrived at by extending these principles.

1) $(X \Lambda Y) < Z$: Simply translates into two constraints $X < Z$ and $Y < Z$. This only requires evaluating a single problem set containing these two conditions in addition to any other precedence relations.

2) $X<(Y\ V\ Z)$: Can be mapped to three possibilities, one with the $X<Y$ constraint, one with the $X<Z$ constraint, and a third with $\{X<Y, X<Z\}$. The answer is the sum of the number of possibilities for $X<Y$ and $X<Z$ **minus** the number of possibilities for $\{X<Y, X<Z\}$ (their intersection). A more efficient, but perhaps less straightforward way to calculate the number of possible orders is:

   orders(X<(Y *V* Z), other constraints) =
   orders(other constraints ONLY)
   - *orders*(Y<X, Z<X, other constraints).

3) $X < (Y \Lambda Z)$: Either task Y or task Z (but not both) may be performed before task X, however, task X must be done before BOTH tasks Y and Z have been completed. Thus, this case is broken down as in case 2) above.

4) $(Y\ V\ Z) < X$: This is also similar to case 2), but with the < and > signs reversed.

5) $X \leq Y$: Requires the evaluation of two possibilities, one with $X<Y$, the other with the X and Y tasks combined into a single task (i.e. they are forced to be performed at the same time). The results of these two cases are then summed up to find the total number of possible orderings.

Just as truth tables grow exponentially with the addition of another variable, for every additional Λ, *V*, or ≤ relationship, the number of simple precedent constraint graphs to be evaluated may increase by a factor of two or three. For example: Assume we have six tasks {A, B, C, D, E, F} and the following set of logical precedence constraints {A<B, A<C, (A *V* D)<E, E≤F}. The logical *V* can create three possible cases, and the ≤ creates another two cases. We can consider all 3*2=6 different graphs and then collect the results to determine the number of possible orders. In this example, the result would be the sum of the following sets of simple precedence conditions:

orders(A<B, A<C, A<E, E=F)
+ *orders*(A<B, A<C, D<E, E=F)
- *orders*(A<B, A<C, A<E, D<E, E=F)
+ *orders*(A<B, A<C, A<E, E<F)
+ *orders*(A<B, A<C, D<E, E<F)
- *orders*(A<B, A<C, A<E, D<E, E<F) =
$30 + 20 - 18 + 72 + 40 - 32 = 112$ orders possible

The third and sixth subproblems each include a violation of the *N-fold* graph assumption and have to be broken down using the graph transformation described in Section 4.7. Thus, the original logical relationships {A<B, A<C, (A *V* D)<E, E≤F} could

ultimately be solved with eight calls to the basic Oracle algorithm. This may seem like a lot of work for such a simple problem, but for a larger number of tasks (with a small number of Λ, V and $\leq$ relations) the superiority of this method over enumeration becomes apparent.

4.3.2. Precedence Graph Representation

Each set of simple precedence constraints corresponding to the general precedence relations can be represented in a single precedence graph. Tasks are represented as vertices and simple precedence constraints are represented as directed edges in the precedence graph. For example, a simple precedence constraint, such as task A must precede task B (A < B), is represented by directed edges from vertex "A" to vertex "B" in the precedence graph.

Initially each vertex in the graph represents a single task. Vertices in the precedence graph are combined into composite vertices as the Oracle algorithm is applied. For N-fold graphs the graph will be reduced into a single vertex which represents all of the original tasks. Each vertex keeps track of how many vertices it represents and how many possible orderings of the represented tasks are possible. The precedence graph is reduced as described in Section 4.4.

4.3.3. N-Fold Graphs

The Oracle algorithm is capable of handling *N-fold* graphs exactly and returns bounding values for non-*N-fold* graphs. Before defining the class of N-fold graphs, a definition for a linear list of vertices is given below.

Definition: linear list of vertices

A *linear list of vertices* consists of n+1 vertices (n≥1) $\{V_0, V_1, \ldots, V_n\}$ and n directed arcs between these vertices, with each vertex V_i having a single incoming directed arc from V_{i-1} and a single outgoing arc to V_{i+1} (i.e. task $V_{i-1} < V_i < V_{i+1}$). V_0 (V_n) may have more than one outgoing (incoming) arc, but V_i (0<i<n) must have only one incoming and one outgoing arc.

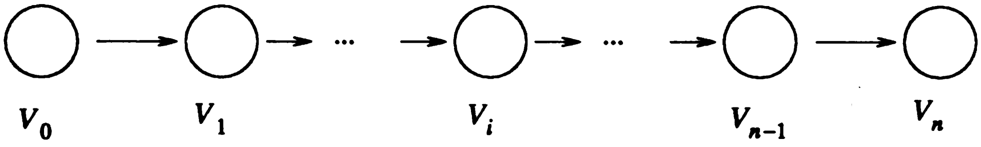


Figure 4-1: A linear list of vertices

With both the *START* and *END* vertices added to a precedence graph (discussed in Section 4.5.2), the Oracle algorithm can exactly determine the number of possible task orderings if this *connected* precedence graph is an *N-fold* graph as defined below:

Definition: N-fold Graph

1) a single vertex is an *N-fold* graph.
2) the result of removing a single vertex, say V_r, in an *N-fold* graph and replacing it with a linear list of 2 vertices ($V_0 \rightarrow V_1$) is an *N-fold* graph. V_0 has as parents the parents of V_r and V_1 has children that were the children of V_r.
3) the result of removing a single vertex in an *N-fold* graph and replacing it with a parallel pattern (as defined in the next section) is an N-fold graph.
4) the result of removing a single vertex in an *N-fold* graph and replacing it with an N pattern (as defined in the next section) is an N-fold graph.
5) nothing else is an N-fold graph.

A few other definitions are in order:

parent vertex - A vertex i will have a vertex j as its parent if their is a directed edge from vertex j to vertex i. Equivalently, the task(s) at vertex j must be performed before the task(s) at vertex i can be performed.

child vertex - A vertex i has a vertex j as its child if a directed edge exists from vertex i to j. The task(s) at vertex i must be performed before the task(s) at vertex j.

ancestors - The ancestors of a vertex i are the parents of vertex i and the parents of those parents, etc. For every ancestor vertex j of i there exists a directed path from j to i.

predecessor vertex - Another term for an ancestor of a vertex.

descendants - Analogous to ancestors, a descendant j of vertex i is a child of i or a child of a child etc. For every descendant vertex j of i there exists a directed path from i to j.

successor vertex - Another term for a descendant of a vertex.

vertices in series - Two or more vertices are said to be in series if they form a linear list of vertices as defined in the previous section.

vertices in parallel - Two or more vertices are considered to be in parallel if they have the same parents and same children vertices.

4.4. Subgraph Replacement

The Oracle algorithm can handle all N-fold graphs exactly by using all three of the subgraph replacements (Figure 4-2): series, parallel, and the N pattern [Mil90b]. The Oracle algorithm replaces occurrences of the three subgraph patterns of vertices as defined in the following subsections with a single composite vertex. Replaced vertices may themselves be composite vertices. By repeatedly reducing a graph using the three subgraph replacement patterns, the Oracle algorithm can reduce all N-fold graphs into a single vertex representing all the original vertices in the graph with the exact number of possible task orderings for all the tasks[11]. During each reduction step the Oracle keeps track of how many tasks have been collapsed into a composite vertex and how many task orderings are possible among all the tasks represented by a single composite vertex. For graphs that cannot be reduced to a single vertex representing all the tasks, an upper and lower bound on the number of possible orderings is computed.

The serial and parallel patterns are the smallest possible subgraphs of two or more vertices that can be used for subgraph reduction. The series and parallel patterns can be used to generate the class of series parallel graphs that appears in the mathematics literature. No unique acyclic subgraphs with non-redundant edges can be made using only three vertices. For example, a linear chain of three vertices $V_1 \rightarrow V_2 \rightarrow V_3$ is not a unique subgraph since it can be made by combining two serial patterns. The N pattern was chosen as a replacement pattern because it was a small unique subgraph and it contains a cross link from vertex $V_1 \rightarrow V_4$ (as in Figure 4-2). This cross link joins two series

[11] The tasks could be assembly, manufacturing, or from any generic task planning problem. e.g. the Oracle algorithm could be used in any problem with tasks and precedence relations - the algorithm is not restricted to the fields of Computer Aided Process Planning or Artificial Intelligence.

patterns that are in parallel with each other. An earlier version of our algorithm contained an X pattern which is an N pattern with an additional edge from V_2 to V_3. However, this X pattern is not unique because it can be reduced using two parallel patterns on (V_1,V_2) and (V_3,V_4) before reducing the resulting composite vertices using the serial pattern.

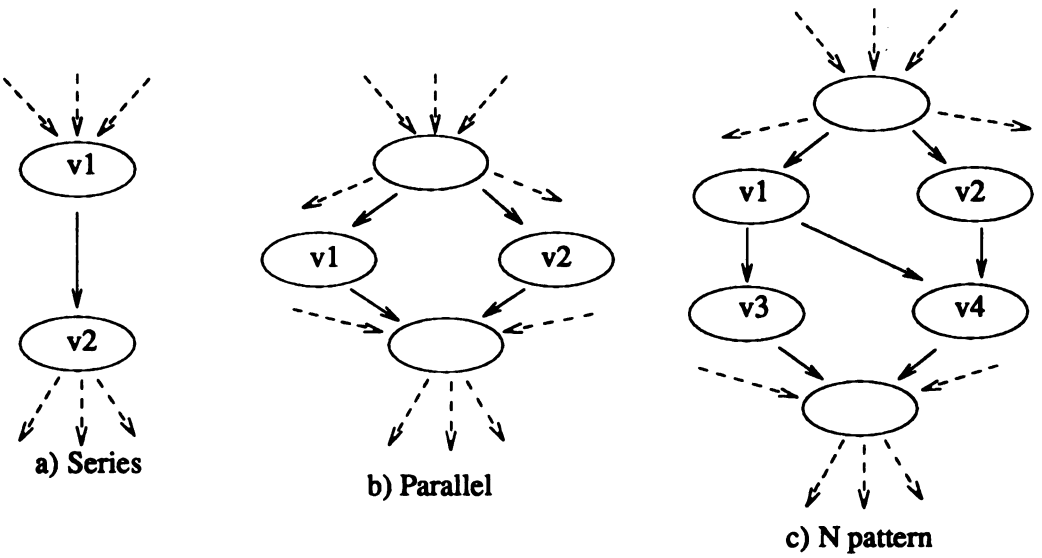


Figure 4-2: The Three Oracle Replacement Patterns

When the precedence graph is created, each vertex represents a single task. To keep track of how many tasks are represented by a single vertex or *composite* vertex, each vertex is assigned two values: 1) N[V_i] for the number of tasks represented by the vertex, and 2) O[V_i] for the number of possible orderings among the tasks represented by this vertex. At the start of the algorithm every vertex i represents a single vertex, so N[V_i] = 1 with a single possible ordering of the one task, and O[V_i] = 1. The graph is reduced by replacing occurrences of the three subgraphs with composite vertices that have values of N[V_c] equal to the number of vertices that have been reduced into the composite vertex, and O[V_c] being the number of possible orderings of the tasks represented by the composite vertex.

4.4.1. Series Replacement

As shown in Figure 4-2 a, two vertices V_1 and V_2 are in series, indicating vertex V_2 must follow vertex V_1. Vertex V_1 must have a single outgoing arc (constraint) to V_2. Vertex V_1 may have many incoming arcs but only one outgoing arc. Similarly, vertex V_2 can only have one incoming arc (from V_1), but may have many outgoing arcs. We can replace vertices V_1 and V_2 in a precedence graph with a composite vertex V_c that has the incoming arcs of V_1 and outgoing arcs of V_2 and the following values for the number of constraints and possible task orderings:

$$\mathrm{N}[V_c] = \mathrm{N}[V_1] + \mathrm{N}[V_2] \qquad \mathrm{O}[V_c] = \mathrm{O}[V_1] * \mathrm{O}[V_2]$$

Clearly, any order represented by V_1 can precede any order represented by V_2, so the combined number of orders is the product of the orders possible for V_1 and V_2.

4.4.2. Parallel Replacement

Figure 4-2 b shows two vertices V_1 and V_2 in parallel[12]. Both vertices must have the same predecessors and successors. The predecessor and successor vertices may have any number of incoming and outgoing arcs unrelated to V_1 and V_2. The tasks represented by vertex V_1 and vertex V_2 can be performed in any combination of orders as long as the task ordering within V_1 and V_2 is not disturbed. We create a composite vertex V_c with the same predecessor and successor vertices as V_1 (V_2) and the number of tasks and task orderings is calculated as:

$$\mathrm{N}[V_c] = \mathrm{N}[V_1] + \mathrm{N}[V_2]$$

$$\mathrm{O}[V_c] = \mathrm{B}(\mathrm{N}[V_1],\mathrm{N}[V_2]) * \mathrm{O}[V_1] * \mathrm{O}[V_2]$$

Note: B(x,y) denotes the binomial of x,y = (x+y)! / (x! * y!).

[12] Although only a single common parent and single common child vertex are shown for the parallel and N replacement patterns, multiple parents and/or children are permitted.

The $O[V_1]$ and $O[V_2]$ terms account for the number of possible orderings within each set of vertices V_1 and V_2. Given two fixed orderings for the vertices represented by V_1 and V_2, the binomial term returns the number of possible combinations of the two sets of ordered vertices [Well72]. The parallel subgraph pattern uses a number of orders formula similar to the W_2 equation that Homem de Mello uses in evaluating the number of task orderings in an AND/OR assembly tree [Home89][13].

4.4.3. N Pattern

The subgraph of vertices V_1, V_2, V_3, and V_4 shown in Figure 4-2 c forms an N pattern in the precedence graph. Note that similar to the parallel case, V_1 and V_2 must have the same predecessors, and V_3 and V_4 must have the same successors. The predecessor and successor vertices may have any number of unrelated incoming and outgoing arcs. Similar to the parallel case, the N pattern can be replaced with a composite vertex V_c with:

$$N[V_c] = N[V_1] + N[V_2] + N[V_3] + N[V_4]$$

1) If $N[V_1]$ is minimum

$$O[V_c] = (B(N[V_4]+N[V_2],N[V_3]+N[V_1]) - \Sigma_{i=1}^{N[V1]} B(N[V_4]-1,N[V_3]+i) * B(N[V_1]-i,N[V_2])) * O[V_1] * O[V_2] * O[V_3] * O[V_4]$$

2) If $N[V_2]$ is minimum

$$O[V_c] = (\Sigma_{i=0}^{N[V2]} B(N[V_1]-1,N[V_2]-i) * B(N[V_3],N[V_4]+i)) * O[V_1] * O[V_2] * O[V_3] * O[V_4]$$

3) If $N[V_3]$ is minimum

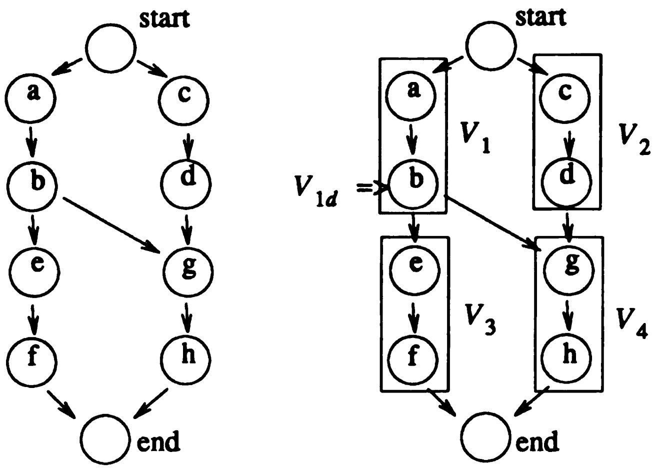
$$O[V_c] = (\Sigma_{i=0}^{N[V3]} B(N[V_4]-1,N[V_3]-i) * B(N[V_2],N[V_1]+i)) * O[V_1] * O[V_2] * O[V_3] * O[V_4]$$

4) If $N[V_4]$ is minimum

$$O[V_c] = (B(N[V_1]+N[V_3],N[V_2]+N[V_4]) - \Sigma_{i=1}^{N[V4]} B(N[V_1]-1,N[V_2]+i) * B(N[V_4]-i,N[V_3])) * O[V_1] * O[V_2] * O[V_3] * O[V_4]$$

The four formulas for $O[V_c]$ are useful in reducing the time complexity of the algorithm by a factor of |V| as discussed in Chapter 5. All of the formulas are equivalent and

[13] Once an *END* vertex has been added to the AND/OR tree the resulting precedence graph can be handled exactly by the series and parallel reduction patterns.



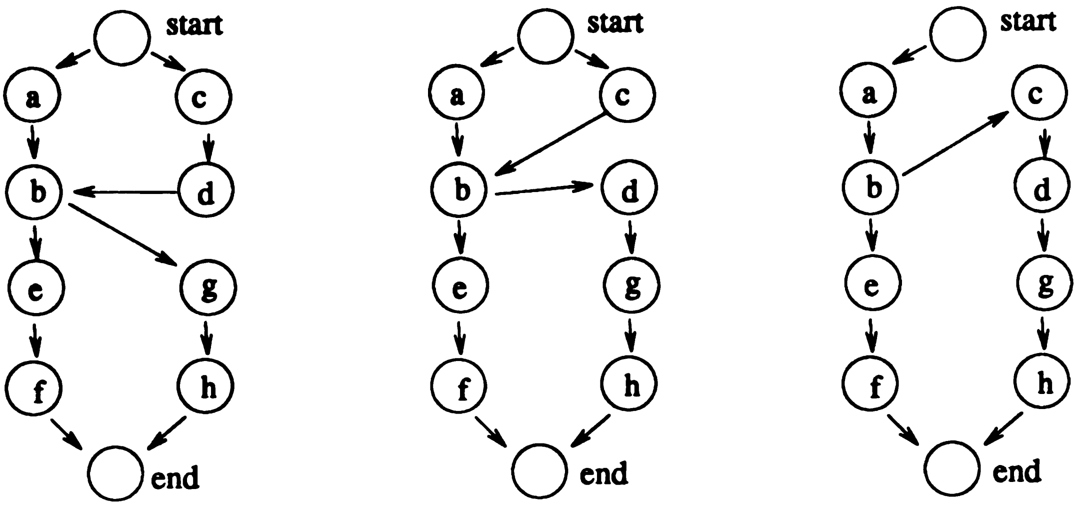
a) N shaped graph

b) $V_{1\text{–}4}$ and the dividing vertex b
A total of 53 Possible Orders

Figure 4-3: Formula 2 for an N shaped graph

are based on the choice of a dividing vertex for the derivation of the number of orderings. Formulas 2 and 3 are similar in that they both directly calculate the number of allowed permutations of the vertices. For example, Figures 4-3 and 4-4 show how formula 2 is applied using the successor vertex (V_{1d}) in V_1 to divide the vertices into two sets of vertices. Figure 4-4 shows all three possible divisions of the vertices using V_{1d} to create two sets of vertices which can each be executed as two parallel lists of vertices. One set of vertices must occur before V_{1d} and the other set of vertices must occur after V_{1d}. With the vertices divided into two sets the number of possible orderings in each set is calculated using the *parallel* binomial formula. The two resulting values are multiplied together since vertex V_{1d} separates these two groups of vertices serially. The N formula sums the number of orders for each of the three subgraphs to obtain 53 possible orders for the graph in 4-3 a. This result requires a total of $|V_2|+1$ calculations for the summation $\Sigma_{i=0}^{N[V2]}$.

In a similar fashion to formula 2, formulas 1 and 4 calculate the number of orders possible for two serial sets of vertices (V_1 and V_3) and (V_2 and V_4) as if they were in



a) 1st Possibility
18 total orders

b) 2nd Possible Division
20 total orders

c) 3rd Possible Division
15 total orders

Figure 4-4: The three possible divisions for Figure 4-3 b

parallel (without the $V_1 \to V_4$ constraint) before subtracting from this value the number of orderings not allowed due to the constraint $V_1 \to V_4$.

4.5. Determining the Number of Task Orderings

The Oracle algorithm works from the set of tasks to be performed and the precedence constraints among these tasks. The set of tasks and precedence constraints is represented as a set of precedence graphs. Each precedence graph is preprocessed as described in Section 4.5.2 and then evaluated using the Oracle subgraph replacement algorithm. The number of possible task orderings for a general set of precedence constraints is the sum of the orders for each precedence graph. Upper and lower bounds are returned for non-*N-fold* graphs. An outline of the method is shown in Section 4.5.3.

4.5.1. Assumptions

1) The set of tasks to be performed and the precedence constraints among the tasks are known.
2) If operations must be performed in parallel they are assigned to a single task (vertex) so they are counted as a single parallel task. To allow for parallel task execution the

≤ relationship could be used between tasks to indicate that the tasks may be performed at the same time or that one task may precede the other.

3) A small number of complex precedence relationships AND, OR, and ≤ are present. (A large number of such complex precedence relationships can create a very large number of precedence graphs to be evaluated.)

4.5.2. Preprocessing a Precedence Graph

The Oracle algorithm assumes that redundant constraints have been removed from each precedence graph. Given the set of tasks {A, B, C} and precedence constraints {A<B, B<C, A<C} the precedence constraint A<C is redundant and must be removed before the Oracle algorithm is invoked. If redundant edges are not removed, then recognizing patterns of series and parallel vertices becomes difficult since redundant constraints change the local graph topology. The series and parallel subgraph patterns cannot be directly applied to graphs with redundant constraints. A vertex with an out-degree of two and a vertex with an in-degree of two are created when vertices in series have a redundant constraint added to them. These serial vertices with an out-degree of two and an in-degree of two can not be reduced using the series reduction pattern. Similarly, adding a redundant constraint from an ancestor of both the parallel vertices to one of the parallel vertices causes the resulting subgraph not to match the parallel reduction pattern.

Precedence graphs may or may not consist of a single *connected* set of vertices. A *START* vertex and an *END* vertex are added to a precedence graph to simplify the Oracle algorithm and the definition of *N-fold* graphs as defined in Section 4.3.3. The *START* vertex must precede all other vertices in the graph. Since precedence relations are transitive, we only need to add a directed arc from the *START* vertex to any other vertex in the graph that does not have any precedence constraints (i.e. all vertices with an in-degree of 0). Similarly, directed edges are added from all vertices with an out-degree of zero to the *END* vertex. This does not change the number of possible orderings for the tasks, but instead insures that the graph is *connected* and can be easily traversed by the Oracle algorithm. All the vertices can be reached by following a directed path from the *START* vertex or alternatively by following reverse edges from the *END* vertex.

4.5.3. The Oracle Procedure is an Algorithm

An overview of the high level steps in the Oracle algorithm is given below followed by a discussion of why these high level steps constitute an *algorithm*. Details of the algorithm, including its time and space complexity, are given in Chapter 5.

1) Represent tasks to be performed as vertices in a graph; edges represent order constraints.
2) Remove redundant constraints (directed edges)
3) Initialize each vertex with N[V_i]=1 and O[V_i]=1.
4) Add a *START* vertex and *END* vertex as described above to insure graph is connected. Set N[*START*] = N[*END*] = 0 and O[*START*] = O[*END*] = 1 so that these additions will not affect the calculated number of task orderings.
5) Apply the three subgraph reductions to the graph until no more can be applied. If the graph has been reduced to a single vertex, then the exact number of orders has been calculated. Otherwise calculate both a lower and upper bound on the number of possible task orders using the remaining non-reducible graph.

When each of the five high level steps are broken down into a series of substeps as described in Chapter 5, all of the substeps are *definite* and *effective*[14] with an overall execution time complexity of $O(|V|^2 * |E|)$. Thus, the Oracle is an algorithm. Not only is the algorithm guaranteed to terminate (due to the $O(|V|^2 * |E|)$ execution time result), but in practice the Lisp implementation of the algorithm returns an answer in a reasonable amount of time as shown in Section 5.6.

4.6. Examples

Five different sets of tasks and precedence constraints illustrate the usefulness of the Oracle algorithm for determining the number of different task orderings possible.

[14] An algorithm consists of a finite number of well defined steps which will always terminate in a finite amount of time for all possible inputs.

4.6.1. Weekday Tasks

Suppose we have eight tasks which are to be performed after waking up in the morning and before leaving for work. Alphabetically the eight tasks are {brush_teeth, eat_breakfast, get_dressed, get_paper, let_dog_in, put_dog_out, shave, shower}. Although many precedence constraints may exist (which may be very difficult to determine automatically), we assume only the following five constraints.

1) shower < shave
2) shower < get_dressed
3) get_paper < eat_breakfast (want to read the paper during breakfast)
4) eat_breakfast < brush_teeth
5) put_dog_out < let_dog_in (dog starts in the house)

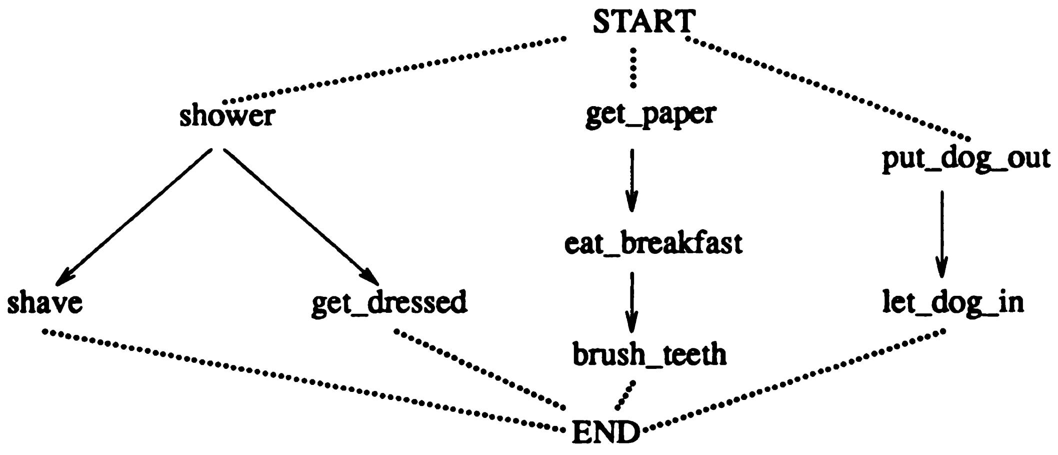


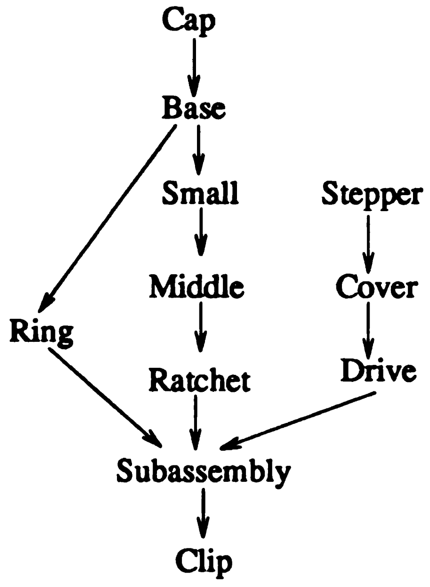
Figure 4-5: Weekday Morning Tasks

The precedence graph for these eight tasks and five precedence constraints is shown in Figure 4-5. A total of 1120 possible orders exist as calculated by the *Oracle* algorithm and verified by an enumeration routine. Although it may be possible to evaluate all 1120 task orderings in detail, adding another set of three generic tasks, say (A, B, C) with the constraints {A<B, B<C}, would cause the number of possible sequences to jump to 184,800. The addition of three such tasks is an example of how the Oracle algorithm can be used as a tool to determine how changing the set of tasks and constraints affects the number of possible task orderings and hence the complexity of the search for an efficient plan.

4.6.2. Gearbox Assembly

Cap < Base
Base < Small Gear
Base < Ring Gear
Small Gear < Middle Gear
Middle Gear < Ratchet Gear
Ratchet Gear < Subassembly
Ring Gear < Subassembly
Stepper Wheel < Cover
Cover < Drive Gear
Drive Gear < Subassembly
Subassembly < Clip

a) Precedence Constraints



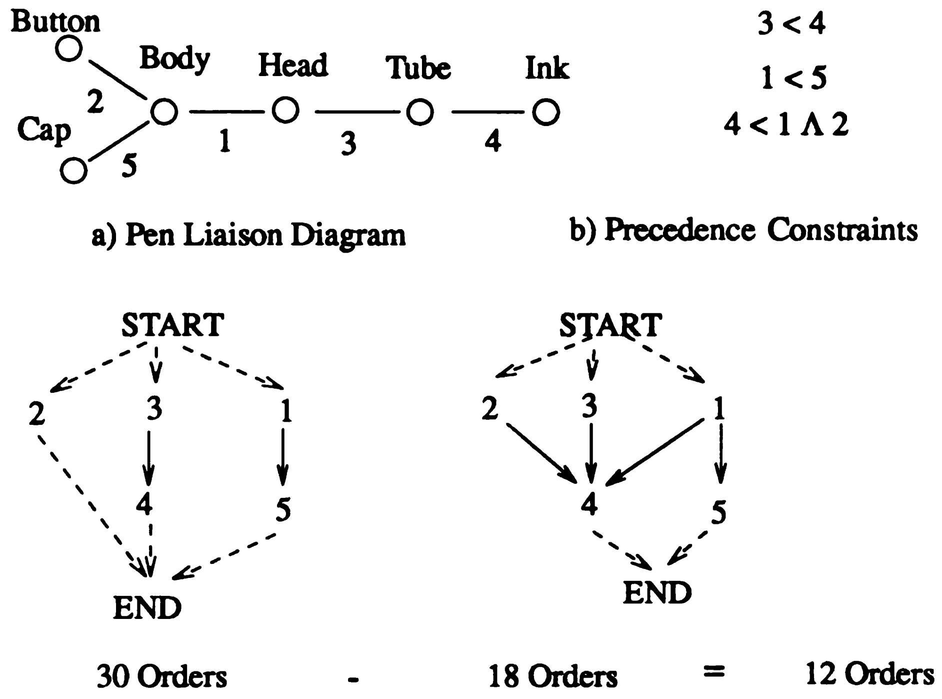
b) Precedence Graph

Figure 4-6: Gear Box Example

This example is from work by Fox and Kempf on opportunistic scheduling in an automated assembly environment [Fox85]. Figure 4-6 a shows the precedence constraints among the individual components in a gearbox assembly. The precedence graph for these precedence constraints is shown in Figure 4-6 b. After *START* and *END* vertices have been added to the precedence graph, the Oracle algorithm can correctly determine that exactly 336 different assembly sequences exist. The Oracle algorithm only needs to apply the series and parallel subgraph patterns to reduce all the representative vertices into a single composite vertex that represents all N[V_c] = 11 assembly tasks with O[V_c]=336 possible task orderings.

4.6.3. Pen Assembly

This example is from Bourjault and illustrates the conversion of complex precedence relations to more than one precedence graph. The liaison diagram of a 'throw



a) Pen Liaison Diagram

b) Precedence Constraints

c) Resulting Precedence Graphs

Figure 4-7: Bourjault Pen Example [Bour84]

away' ball point pen with six different components is shown in Figure 4-7 a [Bour84]. Its three general precedence constraints (Figure 4-7 b) require the evaluation of two precedence graphs to determine that 12 orders exist. The translation from general precedence constraints such as "4 < (1 Λ 2)" to multiple precedence graphs is discussed in Section 4.3.1. The second subgraph requires the application of one parallel pattern (for vertices 2 and 3). Then the N pattern is applied to the combined (2,3) composite vertex and vertices 1, 4, and 5 to find a total of 18 possibilities which must be subtracted from the result of the first precedence graph (30 possible orders) to determine that 12 possibilities exist. The number of possible orderings for this low task count example can be quickly computed using strictly enumerative techniques; however, it is computationally intensive to enumerate loosely constrained assemblies with a larger part count.

4.6.4. Mechanical Mouse Assembly

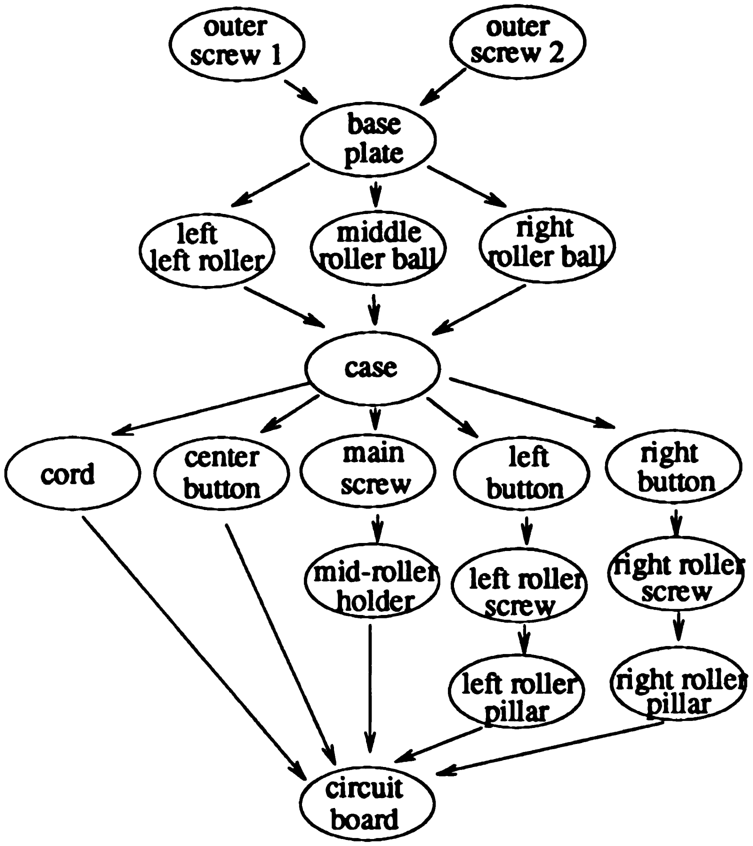


Figure 4-8: Mouse precedence graph (1 of 2)

Precedence graphs for the mechanical mouse described in Chapter 3 are shown in Figures 4-8 and 4-9. The precedence graphs for the mouse assembly of 13 subparts and five fasteners were derived using the following rules for product design for automatic assembly from Boothroyd's book [Boot82]:

1) Ensure that the product has a suitable base part on which to build the assembly.

2) Ensure that the base part has features that will enable it to be readily located in a stable position in the horizontal plane.

3) Build the part up in layer fashion with each part being assembled from above and positively located so that there is no tendency for it to move under the action of horizontal forces during any movement of the partial subassembly / work carrier.

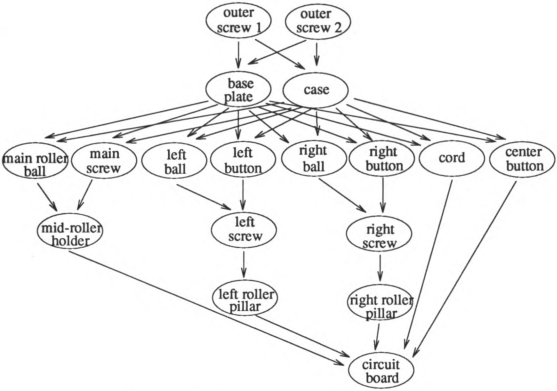


Figure 4-9: Mouse precedence graph (2 of 2)

The circuit board was chosen as the base part[15] for the mouse assembly due to its size and rectangular shape. The mechanical mouse has several screws and components which must be assembled from different directions requiring at least one refixturing step as shown in Appendix II for an efficient assembly sequence called "m3". Due to the large number of vertices in the graph that are in parallel with one another a total of over 58 million (57,657,600 + 604,800) different assembly sequences are possible. Variations of up to 76% in assembly time were shown in Section 3.4.2, indicating the importance of finding an efficient assembly sequence. The large number of feasible assembly orders makes enumeration of assembly sequences impractical. Heuristics, domain knowledge, and/or efficient search techniques are required to insure that only a reasonable number of

[15] The base part is always *assembled* first to serve as a stable building block for the remaining parts in the assembly.

different assembly sequences are actually evaluated in finding an efficient solution.

4.6.5. Door Mechanism Subassembly

Figure 4-10 shows a precedence graph for the assembly of a door knob and lock consisting of 14 parts. Each vertex represents the assembly of one of the 14 parts. The 15 precedence constraints (directed arcs in the graph) could correspond to a subproblem derived from a more complex set of precedence constraints as discussed in Section 4.7. This example does not specifically deal with liaisons in the assembly sequence; however, the Oracle algorithm has been successfully applied to liaison assembly sequences such as DeFazio and Whitney's axle assembly [DeFa87]. The *Oracle* algorithm correctly determined that 308,880 possible orders exist for the assembly of the 14 doorknob parts.

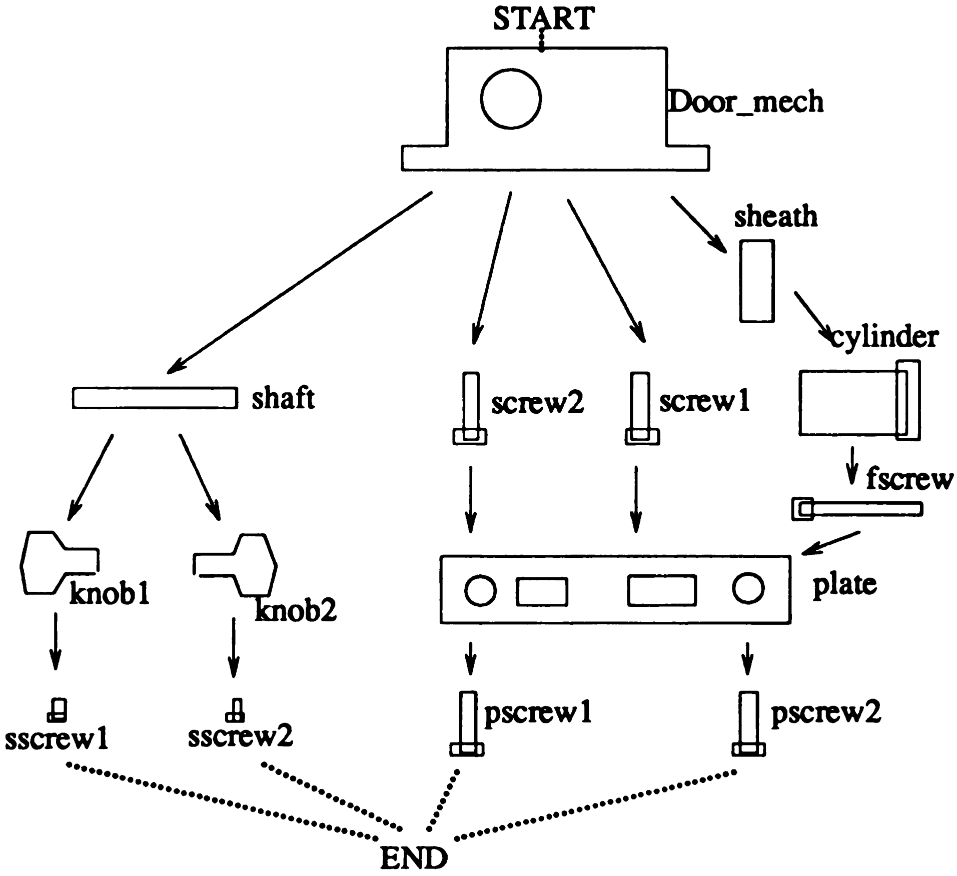


Figure 4-10: One of Several Precedence Graphs for a Door Mechanism

It may seem obvious that the doorknob *subassembly* (sscrew1, knob1, shaft, knob2, sscrew2) could be considered separately from the other door lock parts. Doing this leaves only 40 possible orders for the remaining nine parts, a much more manageable set of alternatives. (Actually, the doorknob *subassembly* cannot be completely assembled separately before being assembled with the door lock, but rather the shaft must be inserted in the door lock mechanism before *both* of the knobs have been assembled.) The Oracle algorithm could be used in a manufacturing design setting by an operator who could propose such a removal of tasks and constraints to see how the change would affect the number of possible assembly orders. Of course, it is possible that an optimal assembly sequence would require an analysis of both the door knob handle *subassembly* and other lock parts. For example, similar tools may be required in both the shaft-knob1 and screw1-door_mech assembly steps. Thus, minimizing tool changes may indicate a mixing of these assembly steps.

4.7. Handling more than N-Fold Graphs

The Oracle algorithm runs quickly on problems that meet the three assumptions given in Section 4.5.1. If the resulting precedence graphs are N-fold graphs an exact value for the number of possible task orderings will be returned. For other graphs upper and lower bounding values on the number of orders are returned. As discussed in Section 5.4, determining how tight these bounds are is difficult. Ideally, we would also like to be able to evaluate non-N-fold graphs exactly. As was mentioned earlier, the general problem of determining the number of linear extensions of a poset is intractable. Therefore, we cannot expect to solve all problems in such a low polynomial time algorithm. However, there are several promising methods for increasing the set of graphs that can be evaluated exactly.

4.7.1. Graph Transformation

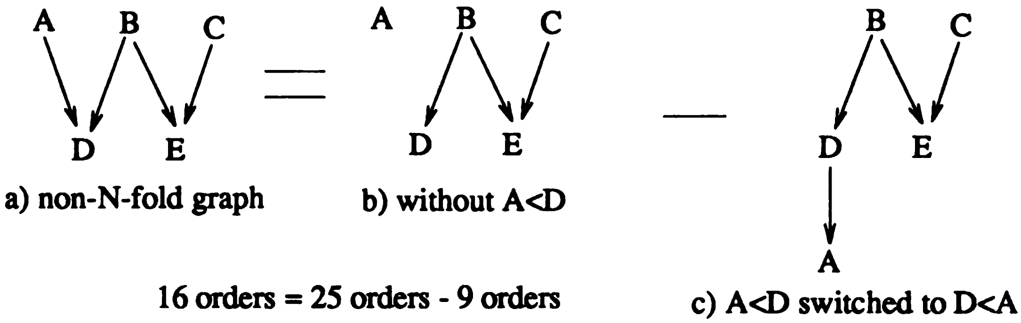
It may be possible to transform non-N-fold graphs into a set of graphs that are N-fold graphs. For example: Figure 4-11 a shows a trivial *W shaped* graph that is not an N-fold graph. The Oracle algorithm described in Chapter 5 cannot precisely determine the proper number of orders because the N-fold assumption is violated.

To solve exactly for the graph in Figure 4-11 a (and other more complex graphs with similar problems), we can make use of the following equality: Given a precedence constraint, i.e. $X<Y$, the number of orders specified by the other constraints and $X<Y$ equals the number of orders without the $X<Y$ constraint **minus** the number of orders allowed by the other constraints and the *reverse of $X<Y$*, i.e., $Y<X$. This can be written as:

$$\textit{orders}(X<Y,\text{other constraints}) = \textit{orders}(\text{other constraints}) - \textit{orders}(Y<X,\text{other constraints})$$

This is true since we do not allow X and Y to be executed at the same time, and the total number of orderings is equal to:

$$\textit{orders}(\text{other constraints ONLY}) = \textit{orders}(X<Y,\text{other constraints}) + \textit{orders}(Y<X,\text{other constraints})$$



a) non-N-fold graph

b) without A<D

c) A<D switched to D<A

16 orders = 25 orders - 9 orders

Figure 4-11: Graph Identity

To find the exact number of possible orders for the graph of Figure 4-11 a, we can intelligently apply the identity described above. The total number of possible orderings allowed by the precedence graph in Figure 4-11 a is the number allowed in Figure 4-11 b **minus** the number allowed in Figure 4-11 c. Note: In Figure 4-11 b the precedence constraint A<D has been dropped, and in 4-11 c it has been reversed to become D<A. Thus, the total number of orders allowed in 4-11 a is as follows:

$$\mathit{orders}(4\text{-}11\ a) = \mathit{orders}(4\text{-}11\ b) - \mathit{orders}(4\text{-}11\ c) = 25 - 9 = 16.$$

Unfortunately, every application of this identity to the graph doubles the number of subproblems that must be solved to find the final answer. Thus, this method is only practical when a small number of constraints are violating the N-fold graph assumption of the Oracle algorithm.

4.7.2. Subgraph Replacement

Subassemblies and hierarchical groups can be helpful in reducing the computational burden. If a precedence graph does not meet the assumptions of the Oracle algorithm it may be possible to identify a subgraph or subgraphs that cause the problem. Cut-sets have been used in the past to identify possible choices for subgraphs. If these subgraphs are not very large, it may be possible to either identify the subgraph as a special known case with a precalculated number of orderings, or to determine the number of possible orders (via enumeration) within that subgraph. Once a problem subgraph has been identified and evaluated, it can be replaced by a *representative* vertex with the appropriate number of possible orderings. This may create a new graph that is an N-fold graph which would allow the Oracle algorithm to find an exact value for the number of orders in the graph.

Node aggregation, conditioning, and the use of cliques have proved helpful in addressing the inference problem for arbitrarily connected networks [Chan89]. Although the node aggregation scheme proposed by Chang and Fung is not directly applicable for use with the Oracle algorithm, some of the techniques such as identifying partitions in a graph appear to be useful. Wells gives an algorithm for the enumeration of poset consistent permutations [Well72]. For small non-N-fold subgraphs, this algorithm could be applied to split the problem into simpler and simpler parts, with the result being the sum of the subproblems. The number of subproblems depends heavily on how many incomparable vertices are associated with the "*cleavage*" vertex.

4.8. Summary of Chapter 4

Determining the number of possible orderings for a set of tasks is helpful in selecting appropriate search and evaluation strategies to be used in finding an optimal solution. A set of |V| tasks may have up to |V|! different orderings depending on the precedence constraints. A set of tasks and precedence relations can be transformed into one or more precedence graphs. Each of these precedence graphs can be evaluated using the Oracle algorithm. The Oracle uses the series, parallel, and N pattern reduction patterns to determine the exact number of orderings for N-fold graphs and produces upper and lower bounds for the number of orderings allowed in non-N-fold graphs. The series, parallel, and N pattern formulas are given, and the correctness of the reduction formulas is discussed. Several examples illustrating the use of precedence graphs in determining the number of possible task orderings are given. The chapter ends with a discussion of several methods for increasing the number of graphs that can be evaluated exactly.

Chapter 5

The Oracle Algorithm

The Oracle algorithm gives advice on the number of different task orderings given a precedence graph. A directed graph representation for the tasks and precedence constraints is processed by a graph reduction algorithm to determine the number of possible task orderings. For N-fold graphs the exact number of possible task orderings can be found using the serial, parallel, and N shaped reduction patterns. If the graph is not an N-fold graph, then it cannot be reduced to a single vertex using the three reduction patterns. Bounding values on the number of possible task orderings are computed for non-N-fold graphs. The Oracle algorithm has an $O(|V|^2 * |E|)$ time complexity and an $O(|V|^2)$ space complexity.

5.1. Representation and Initialization

Precedence graphs are used to represent the set of tasks to be performed and the precedence constraints among the tasks. Section 4.3.1 describes how to map a set of general precedence relations (with AND, OR, and ≤ relations) into sets of simple precedence relations of the form $A < B$. Each set of simple precedence relations can be represented in a single precedence graph G=(V,E). Tasks are represented as vertices in G and a precedence constraint between two tasks is represented by a directed edge between the two representative vertices. *Initially*, each vertex in the graph represents a single task; often in this section the terms vertex and task are used interchangeably. As the Oracle algorithm parses an initial graph, several vertices may be reduced to form one composite vertex, as described below.

For the purpose of determining the time and space requirements of the algorithm we will use |V| to represent the original number of vertices and |E| to represent the original number of directed edges in a single precedence graph G=(V,E). The run time analysis given in this chapter refers to a single precedence graph -- NOT a set of general precedence relations since representation of general precedence relations may require more than one precedence graph.

5.1.1. Graph Initialization

The directed graph is represented using adjacency information for each vertex i ($0 \leq i < |V|$) in the graph. The values stored at each vertex are:

1) N[i], the number of tasks represented by vertex i. Initially, each vertex represents a single task so N[i]=1.
2) O[i], the number of orders possible among the tasks represented by a vertex i. Initially, O[i]=1 since only a single possible task ordering is feasible for the single task represented at a particular vertex.
3) CHILDREN[i], a set of the vertices that can only be performed AFTER this task has been performed. All vertices $0 \leq j <$ |CHILDREN[i]| in CHILDREN[i] have a single directed edge from vertex i to vertex j.
4) PARENTS[i], a set of the vertices that must precede the execution of this vertex. All vertices $0 \leq j <$ |PARENTS[i]| have a single directed arc from vertex j to vertex i.
5) NCHILDREN[i], for the number of children vertices of a vertex. NCHILDREN[i] = |CHILDREN[i]|.
6) NPARENTS[i], for the number of parents of a vertex. NPARENTS[i] = |PARENTS[i]|.

These items make up the precedence graph representation and can be created in O(|V|+|E|) time given a numbered set of tasks $0 \leq i < |V|$ and a set of |E| simple precedence constraints between pairs of tasks. The space requirements are also O(|V|+|E|) since each vertex has a fixed number of values/lists and the total number of possible elements in all the lists is O(|E|) in size.

Using bitmaps[16] to test whether or not two vertices have the same set of children reduces the time complexity of this stage in the algorithm by |V| for $|V| \leq 128$. Each

[16] Bitmaps can also be used in Sections 5.1.3, 5.2.2, 5.3.1, and 5.3.

element of the CHILDREN[] (or PARENTS[]) set can be represented as a bit map of |V| bits with a one in bit location j of CHILDREN[i] indicating that vertex j is a child of vertex i. If the bit maps of two vertices are identical, then the vertices have the same children. When bitmaps are used to store the parents and children information for each vertex the space requirement of this stage becomes $O(|V|^2)$.

5.1.2. Insuring that the Graph is Connected

A *START* vertex and an *END* vertex are added to the precedence graph representation before applying the Oracle algorithm to insure that the graph is *connected* and can be easily traversed. The *START* vertex must precede all other vertices in the graph. Since precedence relations are transitive, we only need to add a directed arc from the *START* vertex to any other vertex with an in-degree of zero. The *START* vertex is an ancestor of every other vertex in the graph. Similarly, a directed edge is added from all vertices with an out-degree of zero to the *END* vertex.

Adding the START and END vertices does not change the number of possible orderings for the tasks since the start vertex MUST precede all other vertices and the END vertex must *follow* all other vertices. All the vertices can be reached by following a directed path from the *START* vertex or alternatively by following reverse edges from the *END* vertex. After the START and END vertices have been added, the number of vertices in the graph is |V|+2. However, to simplify the notation, in the remainder of this chapter we will only refer to |V| vertices in the precedence graph instead of |V| + 2. Since $O(|V|+2) = O(|V|)$ this change in the notation will not affect the complexity analysis.

The START and END vertices require the addition of edges (constraints) being placed between the START or END vertices and some of the original task vertices in the graph. The number of added edges can range from 2 to 2*|V| depending on the precedence graph topology. Since G is connected $|E| \geq |V|-1$, so $O(|E|) \geq O(|V|)$ after this stage in the Oracle algorithm, so O(|E|) = O(|V|+|E|). The addition of the START and END vertices requires O(|V|) time since each original vertex $0 \leq i < |V|$ is checked to see if NPARENTS[i] or NCHILDREN[i] are zero indicating that it is to be added to the CHILDREN[START] or PARENTS[END] lists. This requires an additional O(|V|) space to store the START and END vertex values and lists.

To keep from affecting the calculated number of task orderings the O[START] and O[END] values are set to one. Additionally, the values for N[START] and N[END] are set to 0 since they do not represent one of the original tasks to be performed.

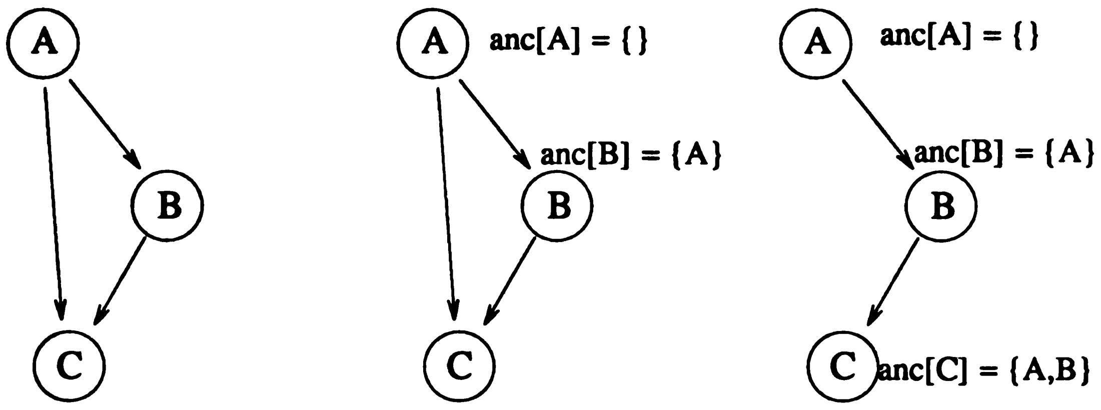
5.1.3. Removing Redundant Constraints

Definition: Redundant edge (constraint)

An edge $e_{i,j}$ from a parent i ($0 \leq i <$ NPARENTS[j]) to vertex j is **redundant** if i is an ancestor of one of the other parents of vertex j. Alternatively, the edge from i to j is redundant if there is another directed path of length greater than one from vertex i to vertex j.

The redundant edges in a precedence graph can be removed leaving a unique set of representative directed edges. There is a unique transitive reduction for every finite acyclic directed graph [Aho72]. Removal of redundant edges reduces the number of edges for all subsequent processing and simplifies the graph so that series and parallel reduction patterns can be easily identified.

Redundant constraints are removed from the precedence graph by visiting each vertex in the graph and determining which, if any, of the incoming directed arcs are redundant. Figure 5-1a shows a graph with three vertices A, B, and C. The directed edge (constraint) from vertex A to vertex C is redundant since a path of length greater than one also exists from A to C through B. The order in which the vertices are evaluated is the same as in a topological sort. A vertex is checked for redundant incoming edges only after all of its ancestors have been checked for redundant edges.



a) Redundant edge A → C b) A & B processed c) Redundant algorithm result

Figure 5-1: Redundant edge removal

The algorithm processes each vertex once and only once by keeping track of the number of unprocessed parents at each vertex. Initially, the number of unprocessed parents:

U_P[i] = NPARENTS[i] for 0 ≤ i < |V|.

Only vertices with U_P[i] = 0 are processed. Processing starts with the START vertex since it is the only vertex with a U_P[] value of 0. When a vertex is processed all of its incoming directed edges are checked to see if they are redundant and the U_P[] value for each child of the processed vertex is decremented by one. When the U_P[] value for a vertex becomes 0 each of its parents have been processed. If unprocessed vertices remain and none of them have a U_P[] value of 0 then a directed cycle exists in the precedence constraints. A directed cycle in the precedence constraints indicates an over-constrained set of tasks that cannot be executed without violating a precedence constraint.

Redundant directed edges can be found by keeping track of the ancestors for each processed vertex and checking to see if any parent vertex is an ancestor of another parent vertex. An edge from a parent j (0 ≤ j < NPARENTS[i]) to vertex i ($e_{j,i}$) is **redundant** if j is an ancestor of one of the other parents of vertex i. The set of ANCESTORS[] at a vertex is the union of all its parents and the ancestors of all its parents. The graph in Figure 5-1b shows that vertex A and B have been evaluated. Vertex C is evaluated last and the edge from A to C is found to be redundant because vertex B has A as an ancestor. Figure 5-1c shows the graph with the redundant constraint removed.

An efficient way to represent the sets of ancestors at each vertex is with bit vectors. Each vertex i (0 < i < |V|-1) in the graph is assigned a vector of |V| bits called ANCESTORS[i]. Each of the b_j bits 0 ≤ j < |V| -1 indicates whether or not each one of the j vertices is an ancestor of vertex i.

The topo-sort processing order of this algorithm insures that each vertex is processed exactly once, and only after all of its ancestor vertices have been processed. When each vertex is processed a list of its ancestors is created from its parents. These ancestor lists are then used in finding redundant edges incident to the vertex being processed. The algorithm is correct and complete since 1) all redundant edges are detected by this method and 2) only redundant edges are detected.

1) A directed edge $e_{j,i}$ from vertex j to i is redundant IF AND ONLY IF another directed path exists from vertex j to another parent of i. This alternative directed path of length ≥ 2 from j to i makes the edge $e_{j,i}$ redundant. When a vertex is processed, each incoming edge $e_{j,i}$ 0 ≤ j < NPARENTS[i] is tested for redundancy by testing all other parents k of i (0 ≤ k < NPARENTS[i] and j ≠ k) to see if j is in the ancestor list of k. Clearly, if the edge $e_{j,i}$ is redundant, then one of the other parents k will have j in its ancestor list.

2) False positives will not occur for non-redundant edges. If an edge $e_{j,i}$ is not redundant then by the definition of a redundant edge none of the other parents k (0 ≤ k < NPARENTS[i] and j ≠ k) will have j as an ancestor. If j were an ancestor to one to the other parents k, then a path of length ≥ 2 would exist from j to k to i causing an edge $e_{j,i}$ to be redundant. Since j will not appear as an ancestor of any other parent

k, then the edge $e_{j,i}$ cannot be incorrectly identified as being redundant.

The time complexity of this algorithm for removing redundant edges is $O(|V| * |E|)$ since each edge is traversed a fixed number of times and $O(|V|)$ comparisons take place in combining the ancestor lists for each incoming edge. Detecting the redundant edges once the ancestor lists have been created also requires $O(|V| * |E|)$ time since each incoming edge is tested for redundancy using the ancestor lists of the other parents of a given vertex. The space complexity is $O(|V|^2 + |E|)$ to store the vertex ANCESTORS[i] lists and the other edge data.

An optimal algorithm for the general problem of redundant edge detection/ removal requires $O(|V|^{\log_2 7})$ time [Aho72]. However, for a small number of edges ($|E| \approx |V|$), an $O(|V| * |E|)$ time approach, such as the one given here, is more appropriate. Actually, the algorithm approaches $O(|E|)$ time if a small number of redundant edges exist and the number of vertices ($|V|$) is less than the machine word size. This assumes that bit vectors and bit vector operations are used to reduce the complexity by a factor of $O(|V|)$.

5.1.4. Uniqueness of the Reduction Patterns and Resulting Answer

An important fact about the three merging patterns used by the Oracle algorithm (Figures 5-2 and 5-4) is that they are all ***unique*** or ***prime***. That is, no combination of any two patterns can create the third pattern.

Theorem 5.1: The series, parallel, and N patterns are all unique.

The series and parallel patterns are known to be unique [Habi87], so it is only necessary to show that the N pattern is unique with respect to the series and parallel patterns. A close look at the specific definition of each of these three primitive patterns confirms this. The series pattern $V_1 \rightarrow V_2$ is such that V_1 only has a single child, V_2, and V_2 has only V_1 as a parent. Looking at the four vertices in the N pattern (Figure 5-4) shows that no two vertices sharing an edge have this property so no two vertices in the N pattern could be reduced using the series pattern. No two vertices in the N pattern meet the definition of a parallel pattern by having the same set of parents and children, thus, the N pattern is unique. It is impossible to create any one of the three reduction patterns by using a combination of the other two reduction patterns as building blocks. If we consider any node of any graph at any time, that node can be involved in at most one reduction pattern.

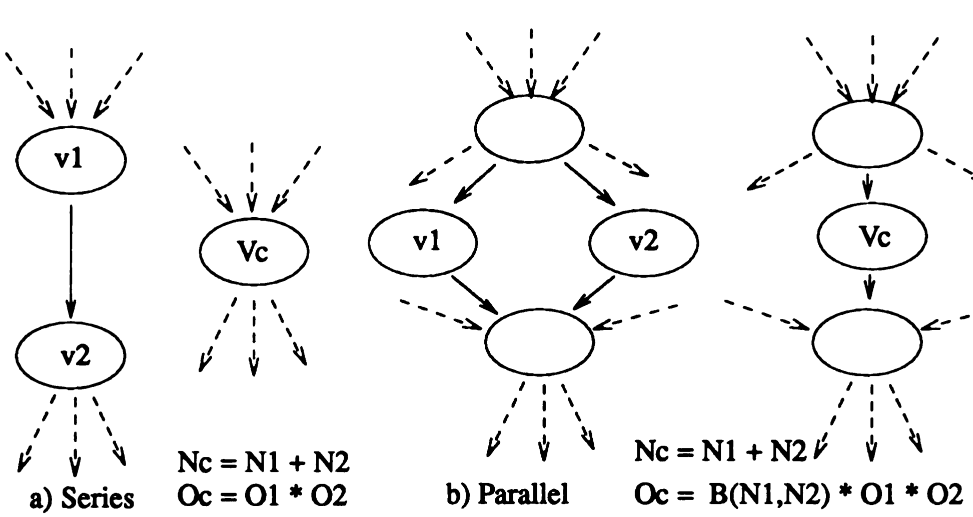
Theorem 5.2: Different sequences of series, parallel, or N pattern reductions on a graph produce the same final result for the number of possible task orderings.

Since each reduction pattern is unique, if a given precedence graph was reduced using two series merges, three parallel merges and one N pattern merge then every possible reduction of that graph will consist of two series, three parallel, and one N pattern reductions. Although it is impossible for the three patterns to be *mapped* to one another, it is possible that a graph may be reduced using different sequential orderings of the same sets of reduction patterns. For example, in a linear chain of three vertices $V_1 \rightarrow V_2 \rightarrow V_3$ it is possible to merge V_1 and V_2 first, or to merge V_2 and V_3 first. In either case, two series merges will be performed so it must be shown that the number of orders will always be calculated to be the same.

Only the series and parallel formulas may operate on common vertices in different orders (as in the previous example). N patterns may not share the same vertices unless they are *nested*, in which case a single possible ordering is forced with the innermost N pattern being reduced first. The series and parallel formulas insure that the answer is always the same regardless of the order in which the vertices are reduced. This is true since addition, multiplication, and the binomial formula are commutative and associative. Combining serial vertices in different orders as in the previous example of $V_1 \rightarrow V_2 \rightarrow V_3$ produces $(O(V_1) * O(V_2)) * O(V_3)$ or $O(V_1) * (O(V_2) * O(V_3))$ depending on the order in which the serial vertices are reduced. Obviously, the resulting values are the same due to the commutative property of multiplication.

5.2. Serial and Parallel Reduction Stage

The first graph reduction stage replaces all the serial vertices by a single vertex and merges all parallel vertex patterns into a single composite vertex (which may then enable other merges). The algorithm uses a READY[] list to keep track of which vertices can be processed next. Each original vertex is processed only once. However, after a parallel combination occurs the resulting composite vertex is also visited. The order in which the vertices are visited is the same as that of a topo-sort algorithm. This algorithm assumes that all redundant constraints have been removed as described in Section 5.1.3.



a) Series

b) Parallel

Figure 5-2: Series and Parallel Vertices

This stage reduces all series and parallel graph constructs. Figure 5-2 shows two vertices in series and two vertices in parallel.

1) Initialization
   a) Create an array U_P[] to represent how many Unevaluated Parents exist for each vertex. This is initialized to be the number of parents of each vertex (NPARENTS[i]).
   b) Create an array called MEET[] which is initially an empty list (NIL) for all i vertices. The MEET[] array of lists is used to keep track of vertices that may be in parallel. MEET[i] lists may be non-nil during this stage IF and only IF more than a single parent exists for vertex i (0≤ i -1).

```
  c) Start checking for serial/parallel patterns at the START vertex. Set READY[] =
     START
2) Do steps a-c while the READY[] list is NON-EMPTY
  a) Take a vertex off of the READY[] list and call it NEXT-V. Set READY[] to be
     READY[] without the NEXT-V vertex.
  b) If NEXT-V has a single parent (NPARENTS[NEXT-V] = 1) and that parent has a
     single child (NCHILDREN[parent of NEXT-V] = 1) then NEXT-V and
     PARENT-NEXT-V are in series and can be reduced into a single vertex using
     SERIES_REDUCE(PARENT-NEXT-V,NEXT-V); otherwise continue on with
     step c).
  c) For each CHILD in CHILDREN[NEXT-V] do i - ii
    i)  Decrement the value of U_P[CHILD] by one.
    ii) If NPARENTS[CHILD] > 1 then a parallel merging MAY be possible Call
        CHECK_PARALLEL(NEXT-V,CHILD)
        Else if U_P[CHILD] = 0 then add CHILD to the READY[] list.
```

Series reduction requires $O(|V| + |E|) \approx O(|E|)$ time since each vertex and edge is visited/looked at a fixed number of times and a single series reduction step takes $O(1)$ time. The CHECK_PARALLEL() algorithm requires $O(|V|^2 * |E|)$ time in the worst-case since it may be necessary to perform $O(|V| * |E|)$ comparisons with a worst-case time complexity of $O(|V|)$ time per comparison. On average, the number of compares necessary will be closer to $O(|E|)$ since most vertices will have a small (nearly constant) number of other incoming edges to be checked. Additionally, the $O(|V|)$ time per comparison will be closer to $O(1)$ as discussed in the Section 5.2.3 leaving a typical time complexity closer to $O(|E|)$. The space complexity is $O(|V| * |E|)$ since each vertex has a merge list. The whole graph will have at most E such merge lists of $O(|V|)$ size each.

5.2.1. Merging of Two Vertices in Series

Algorithm: SERIES_REDUCE(PARENT-NEXT-V,NEXT-V)

This algorithm performs a series reduction on the vertices PARENT-NEXT-V and NEXT-V vertices in the precedence graph representation. Figure 5-2 a shows two vertices in series. Steps 1 and 2 calculate the number of tasks represented by the resulting combined vertex and the number of possible task orderings allowed among the tasks at the combined vertex. The NEXT-V vertex is effectively combined with the PARENT-NEXT-V vertex to produce a composite vertex which replaces PARENT-NEXT-V in the precedence graph. The CHILDREN[] and PARENTS[] pointers are modified to 'remove' NEXT-V from the precedence graph after the serial merge has been performed.

```
1) N[PARENT-NEXT-V]=N[PARENT-NEXT-V] + N[NEXT-V]
2) O[PARENT-NEXT-V]=O[PARENT-NEXT-V] * O[NEXT-V]
        Steps 3 and 4 effectively delete the merged NEXT-V vertex from the pre-
        cedence graph. The vertex PARENT-NEXT-V will then represent both of
        these combined vertices.
3) CHILDREN[PARENT-NEXT-V] = CHILDREN[NEXT-V]
   NCHILDREN[PARENT-NEXT-V] = NCHILDREN[NEXT-V]
4) For each CHILD in the new CHILDREN[PARENT-NEXT-V] list DO
        Substitute PARENT-NEXT-V in for NEXT-V in the PARENTS[CHILD] list.
```

The series reduction code requires a constant amount of time to replace two serial vertices by a single vertex. Its time complexity is O(1) and no additional space is required since the composite vertex is stored in the location where the NEXT-V vertex was stored.

5.2.2. Checking for Two Vertices in Parallel

Algorithm: CHECK_PARALLEL(NEXT-V,CHILD)

This algorithm checks to see if the NEXT-V vertex can be merged in parallel with one of the other parents of the CHILD vertex. If no merge of NEXT-V with another parent of CHILD is possible, this routine adds a sublist of merging information to the list MEET[CHILD]. Parallel merges are performed when detected and then the combined vertex is placed on the READY[] list for further evaluation.

When this algorithm is invoked the NEXT-V vertex is one of several vertices in the graph that have the CHILD vertex as one of their children. Thus, the NEXT-V vertex might be merged via a parallel reduction with one of the other parents of CHILD. The vertex NEXT-V can be merged with one of the other parents of CHILD (say OTHER-

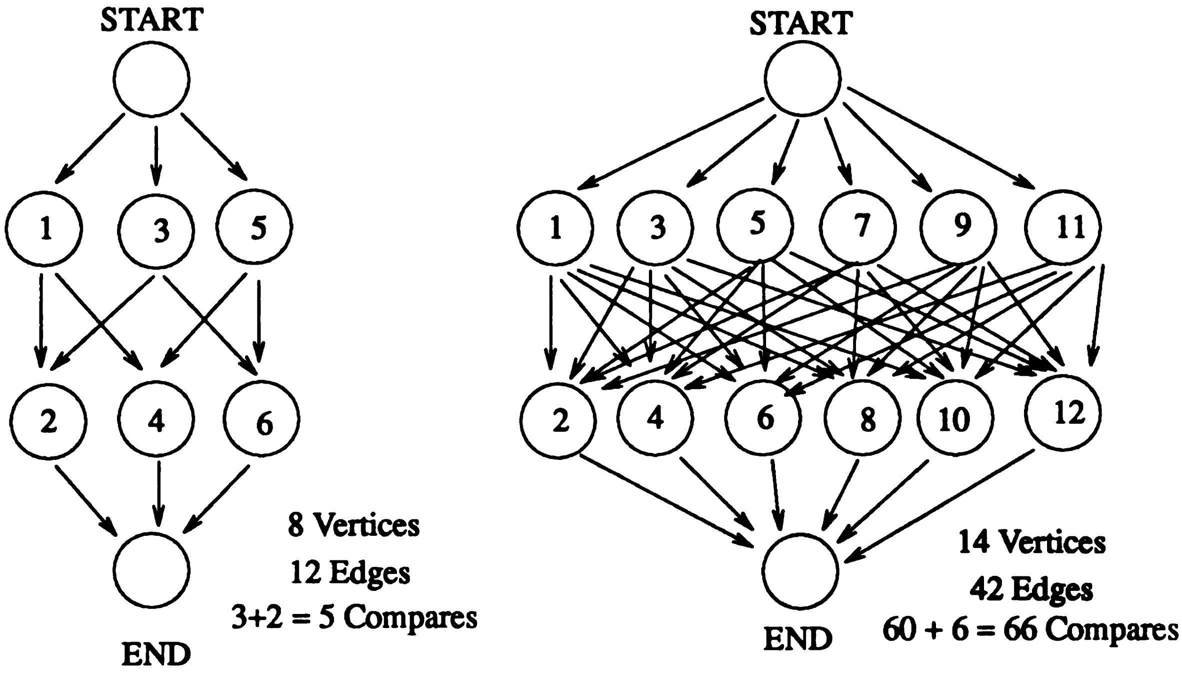
PARENT) IF and ONLY IF NEXT-V and OTHER-PARENT have the same parent and children vertices. In order to detect possible parallel merges this algorithm uses the information stored in the MEET[] list at the CHILD vertex. The MEET[CHILD] list may be NIL indicating that the other parents of CHILD have not yet been evaluated, or MEET[CHILD] may contain a sublist of information for each parent of the CHILD vertex that has been evaluated, but not merged with another parent of CHILD.

The sublist information of each parent that has been visited will consist of three parts. For example the sublist for the NEXT-V vertex is ((PARENTS[NEXT-V]) (CHILDREN[NEXT-V]) (NEXT-V)). In order for two vertices to be in 'parallel' with each other they must have identical PARENTS[] and CHILDREN[] lists. Note: The PARENTS[] and CHILDREN[] lists could be sorted during the REDUNDANT EDGE REMOVAL STEP (by increasing index order) to make this comparison relatively efficient, or bit maps could be used to simplify this equality test. Note: At this point all vertices in series directly above and below the NEXT-V vertex will have been reduced.

1) Set PARALLEL-FOUND = NIL. No parallel vertices have been found that can be merged.
2) For each sublist in MEET[CHILD] check to see if the PARENTS[NEXT-V] and CHILDREN[NEXT-V] are exactly the same as the PARENTS[] and CHILDREN[] lists in the other MEET[CHILD] sublists. If a match is found, then set PARALLEL-FOUND equal to that sublists 3rd element (i.e. the index of the vertex that NEXT-V will be merged with) and go on to Step 3. If all the sublists in MEET[CHILD] have been checked and no match is found then continue with Step 4.
3) Call the parallel merge algorithm to merge the NEXT-V and PARALLEL-FOUND vertex. PARALLEL_MERGE(NEXT-V,PARALLEL-FOUND). Place NEXT-V back on the READY[] list since it is now a composite vertex (and some serial or more parallel merges may now be possible). Increment U_P[CHILD] by one since NEXT-V was placed back on the ready list for reconsideration.
4) Add a sublist to MEET[CHILD] containing (PARENTS[NEXT-V] CHILDREN[NEXT-V] NEXT-V) to be used in future parallel merge checking steps. IF NO MERGE WAS possible and U_P[CHILD]=0 then add CHILD to READY[].

Each call to CHECK_PARALLEL() has a time complexity based on the local graph topology and how many processed, yet unmerged sibling vertices of NEXT-V have saved their merging sublists in the MEET[CHILD] list. Over all possible O(|E|) calls to

CHECK_PARALLEL the total number of possible checks is $O(|V|*|E|)$. If bit maps are used to test for similar PARENTS[] and CHILDREN[] values for NEXT-V and its siblings then the amount of time spent on each check is $O(1)$ for $|V| \leq$ machine word size. However, in general the time is $O(|V|)$ giving a worst-case complexity of $O(|V|^2 * |E|)$ as illustrated in Figure 5-3. Note that most cases will require closer to $O(|E|)$ calls to CHECK_PARALLEL and only a small (nearly constant) number of other siblings will need to be checked. In addition, the time for each comparison can usually be made in $O(1)$ time using the NCHILDREN[] and NPARENTS[] values to screen for obviously non-parallel vertex pairs.



a) 6 Task Case

b) 12 Task Example

Figure 5-3: Tough cases for the CHECK_PARALLEL algorithm

5.2.3. Merging of Two Parallel Vertices

Algorithm: PARALLEL_MERGE(V1,V2)

The parallel merge algorithm PARALLEL_MERGE(V1,V2) merges the tasks represented by vertex V1 and V2 and combines them into a single composite vertex V1 (i.e. overwrites the old V1 vertex information to save space). Figure 5-2 b shows two vertices in parallel. The number of tasks and the number of task orderings represented by V1 and V2 is combined and the results stored in V1. Vertex V2 is effectively removed from the precedence graph by deleting references to V2 from its parents and children vertices.

```
1)  N[V1] = N[V1] + N[V2]
2)  O[V1] = B([N[V1],N[V2]]) * O[V1] * O[V2]
    Where B(x,y) is the binomial of x,y = (x+y)! / (x! * y!)
Now remove references to the merged V2 vertex.
3)  For each PARENT in PARENTS[V2]
        Remove V2 from CHILDREN[PARENT]
4)  For each CHILD in CHILDREN[V2]
        Remove V2 from PARENTS[CHILD]
```

The parallel merge code requires time proportional to the number of incoming and outgoing edges to replace two vertices in parallel by a single vertex. Its time complexity is O(|E|) for all parallel merges and no additional space is required since the composite vertex is stored in the location where the V2 vertex was stored.

5.3. N Reduction Pass

If the graph can be reduced using only series and parallel reductions as described in Section 5.2 then the graph will have been reduced into a single composite vertex with the correct number of orders O[] stored at the remaining vertex. If this is the case, we exit with the exact number of possible orderings known. Otherwise, we attempt to reduce the remaining vertices using the N pattern described in this section. After an N pattern is applied it is possible that further series or parallel reductions may be possible.

Figure 5-4 shows an N type pattern. Note that vertex V4 has exactly two parents; this fact is used in searching for N patterns in the graph. To reduce N patterns the following steps are performed.

```
1) Set CHECK-N-LIST to be empty (NIL).
2) For every remaining vertex I in the graph
      If NPARENTS[I] > 1 and MEET[I] <> NIL then add I to CHECK-N-LIST
```

At this point all the vertices with more than one uncombined parent will be on the CHECK-N-LIST. These CHECK-N-LIST vertices in the graph are candidate positions for N type patterns of vertices.

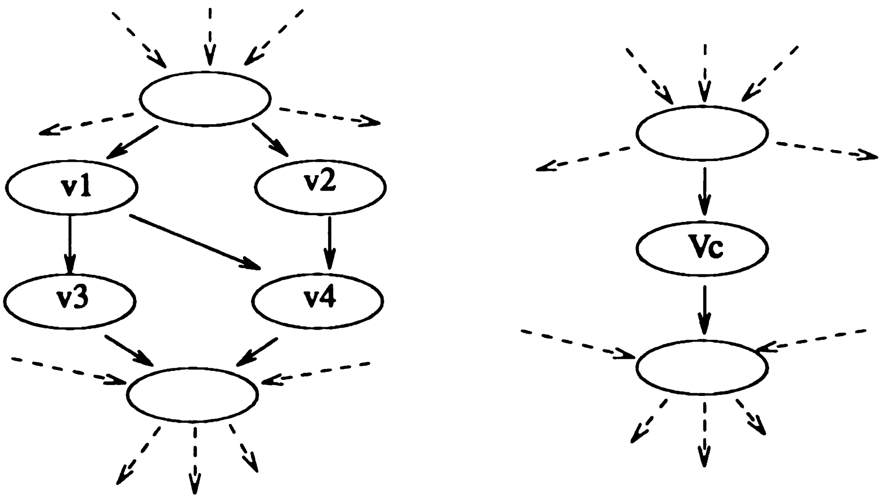


Figure 5-4: An N shaped pattern of vertices

```
3) Set FOUND-ONE = TRUE
4) DO WHILE FOUND-ONE = TRUE
   a) FOUND-ONE = FALSE
   b) FOR each I in CHECK-N-LIST do i-ii
      i) N_TEST(I) to see if I corresponds to a 'V4' vertex in an N pattern.
            If an N pattern is found - it is reduced by N_TEST() into a single vertex.
            N_TEST() then performs a local check to see if any serial or parallel reductions
            are possible. All such local series and parallel reductions are performed before
            continuing.
      ii) If N_TEST(I) was true set FOUND-ONE = TRUE and REMOVE I from
          CHECK-N-LIST
   c) IF FOUND-ONE = TRUE then start over with a) OTHERWISE EXIT
```

This stage reduces all N patterns into single vertices and after each N-reduction it performs all possible parallel and serial reductions before continuing. It would be possible to check for several other patterns besides the N pattern in a similar manner if desired.

Steps 1 and 2 of this N checking algorithm require O(|V|) time and space since each vertex is checked whether or not it is a possible candidate for the location of a V3 vertex in an N pattern. The CHECK-N-LIST will have O(|V|) entries, at most one for each vertex except the START vertex. Steps 3 and 4 will be re-executed on unmerged possible V4 vertices if an N pattern is reduced during a single pass since it is possible that nested N patterns may exist and an N pattern can only be detected and reduced IF all the N patterns contained *inside* of it have been reduced. It is highly unlikely that many nested N patterns will occur in set of say 20 tasks since approximately 4 vertices/tasks are required for an N pattern. Step 4 may be executed $O(|V|^2)$ times if multiple passes through the set of CHECK-N-LIST data are needed. Note: This time complexity could be reduced to O(|V|) with some preprocessing to process the more deeply nested N patterns before trying to reduce the outer N patterns. However, reducing the complexity of this step will not reduce the overall complexity of the Oracle algorithm. Each one of the vertices on the CHECK-N-LIST are checked with the N_TEST() algorithm giving a worst-case complexity of $O(V^3)$ since each N_TEST() algorithm requires at most O(|V|) time to check for similar PARENT[] and CHILDREN[] vertices.

5.3.1. Checking for N Patterns in the Graph

Algorithm: N_TEST(CHECK-VERTEX)

This algorithm performs a check to see if CHECK-VERTEX is a vertex corresponding to vertex V4 in Figure 5-4. If an N type pattern is found it calls REDUCE_N_PATTERN(V1,V2,V3,V4) to reduce the N pattern into a single vertex.

REDUCE_N_PATTERN() also checks the graph locally around the reduced composite vertex to see if any parallel or series reductions can be performed. The variables V1 through V4 refer to labeled vertices in Figure 5-4.

1) V4 = CHECK-VERTEX(assume this is true)
2) If NPARENTS[V4] <> 2 then EXIT (no N type found).
3) At this point two parents of V4 exist. One parent MUST have two children, the other parent MUST have one child (V4). IF NOT then EXIT (not an N-type).
4) V1 = the parent vertex with two children.
   V2 = the parent vertex with a single child (V4).
   If PARENTS[V1] <> PARENTS[V2] then EXIT (not an N-pattern).
5) V1 has two children, one being V4. Call the other child V3.
   If NPARENTS[V3] <> 1 then EXIT (no N pattern found).
   If CHILDREN[V3] <> CHILDREN[V4] then EXIT (not an N pattern).
6) The vertices V1, V2, V3, and V4 make up an N-pattern so now we call the REDUCE_N_PATTERN algorithm with REDUCE_N_PATTERN(V1,V2,V3,V4).

The algorithm requires at most O(|V|) time for each call since the algorithm may have to compare the PARENTS[] and CHILDREN[] data to determine if an N pattern exists and to merge the N pattern into a single vertex.

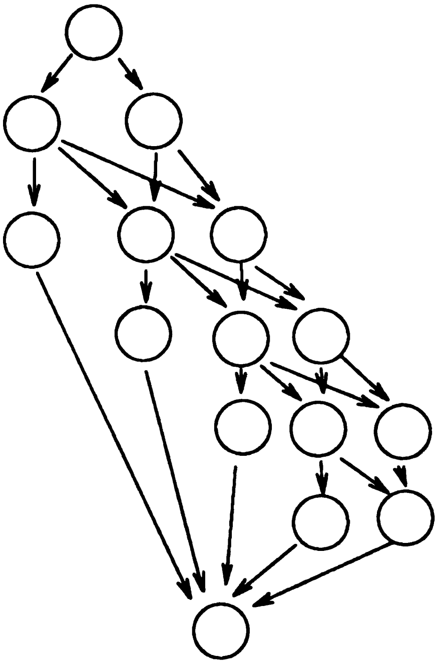
5.3.2. Merging an N Pattern of Vertices

REDUCE_N_PATTERN(V1,V2,V3,V4)

This algorithm reduces the N pattern specified by vertex indices V1, V2, V3, and V4 into a single vertex. It then calls the CLEAN_PARALLEL_SERIAL() algorithm to recursively merge parallel and series patterns that can be merged after the N pattern was reduced to a single vertex. The four vertices V1, V2, V3 and V4 are reduced into a single composite vertex which is stored at vertex V4's position in the graph. References to V1, V2, and V3 are removed from the precedence graph so that the precedence graph reflects the merge operation.

1) N[V4] = N[V1] + N[V2] + N[V3] + N[V4]
2) In order to keep the worst-case calculation complexity from becoming $O(N^2)$ we use one of the four order formulas given in Section 4.4.3 to keep the complexity at O(|V|) for this calculation step (over all N type reductions -- not just a single

```
   application).
3) Now we change the precedence graph so that references to V1, V2 and V3 are
   removed while V4 takes their place.
   a) PARENTS[V4] = PARENTS[V1]
   b) For each INDEX in PARENTS[V4]
          Remove V1 and V2 from CHILDREN[INDEX]
          Add V4 to CHILDREN[INDEX]
   c) For each INDEX in CHILDREN[V4]
          Remove V3 from PARENTS[INDEX]
4) Call CLEAN_PARALLEL_SERIAL(V4) to clean up any parallel and series patterns
   that now may be able to be reduced in the local vicinity of the V4 vertex.
```



Four Nested N Patterns

Figure 5-5: O($|V|^2$) tests are not required using the test in step 2

This algorithm requires at most O(|V|) time to merge all N patterns of vertices. The O(|V|) complexity is due to the O[] formulas used in in step 2 which iterates from i=0 to min(|V1|, |V2|, |V3|, |V4|). By iterating over the smallest set of vertices the overall complexity of this step is reduced to O(|V|) instead of O($|V|^2$) for all N-reductions in the graph which could occur in Figure 5-5 if only a single formula were used.

5.3.3. Checking for Further Parallel/Series Merges

Algorithm: CLEAN_PARALLEL_SERIAL(VERTEX)

This algorithm recursively reduces parallel or series patterns that can now be reduced as a result of an N pattern reduction being performed.

```
1) REDUCTION-FOUND = TRUE
2) While REDUCTION-FOUND = TRUE do
  a) REDUCTION-FOUND = FALSE
  b) CHECK-PARALLEL(VERTEX)
     IF any parallel reductions were performed set REDUCTION-FOUND = TRUE
  c) IF VERTEX has a single parent (NPARENTS[VERTEX] = 1) AND the parent of
     that vertex has a single child (NCHILDREN[parent of VERTEX] = 1) then VER-
     TEX and PARENT-VERTEX are in series and can be reduced into a single vertex
     using SERIES_REDUCE(PARENT-VERTEX,VERTEX) otherwise GOTO step a).
     Set:
          REDUCTION-FOUND = TRUE.
          VERTEX = PARENT-VERTEX
```

This algorithm requires O(|V|) time for the while loop in step 2. This loop is executed at most O(|V|) times since it loops through only if a successful merge for a series or parallel vertex combination occurred. Each series of parallel reduction reduces the number of vertices in the graph by one each time through the loop so at most O(|V|) passes may occur. Each pass may require O(V) comparisons of parent/children vertices giving rise to $O(|V|^2)$ time complexity. No additional space is required.

5.4. Upper and Lower Bound Estimates

If the graph has been reduced into a single composite vertex via the application of the previous stages we can exit with the exact number of possible orders being the O[] value of the remaining vertex. However, if this is not the case, then a LOWER and UPPER bound on the number of possible task orderings is calculated as described in this section. These calculations are performed by two separate simplifying algorithms, one which always produces a lower bound on the number of possible task orderings, and the other which always produces an upper bound on the number of task orderings.

Both the Lower and Upper bound algorithms use the precedence graph that was created by the application of the N pattern reduction stage. For both the lower and upper bounds the graph is simplified using certain heuristics which are guaranteed to underestimate or overestimate the number of possible task orderings. Several different heuristics may be applied depending on the graph topology. If the resulting simplified graph can be reduced using the series, parallel, and N type patterns then the algorithm exits with this final value. However, if even the simplified graph cannot be reduced exactly, then the algorithm makes a final pass reducing all the remaining vertices either serially for a lower bound or using the parallel reduction pattern to obtain an upper bound on the number of possible task orderings.

5.4.1. Lower Bound

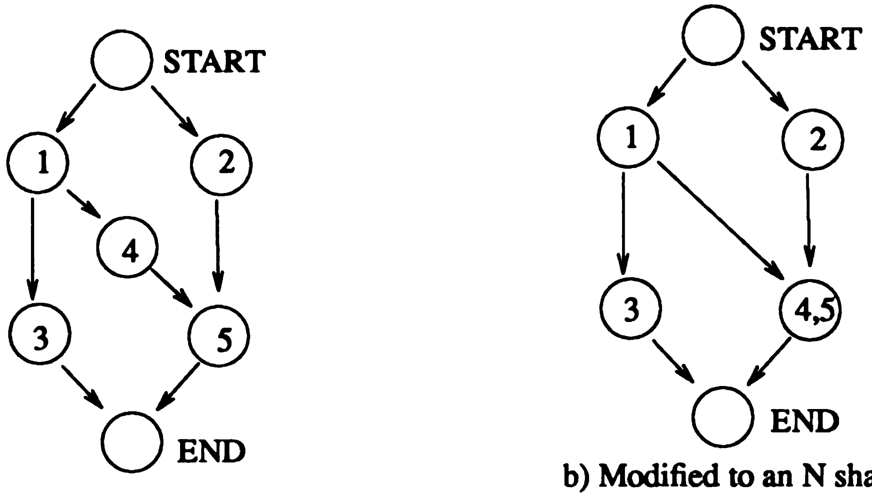
In order to find a lower bound on the number of task orders the LOWER_BOUND() algorithm simplifies the precedence graph using heuristics that can only produce a lower bound on the number of orders. These heuristics were developed by the author for application to a precedence graph in the hopes that the resulting simplified graph can be evaluated exactly using the series, parallel and N pattern reductions. Even after the simplifications have been applied it is possible that the remaining graph cannot be evaluated exactly. In this case we make the simplifying assumption that the remaining vertices are all in series. Combining the remaining vertices as if they were in series gives a lower bound on the number of possible task orderings and will reduce any remaining vertices into a single vertex containing a lower bound on the number of possible orderings.

The lower bound algorithm is applied to each *parallel subgraph* in the reduced precedence graph that results after the series, parallel, and N patterns have been applied to the original precedence graph. If the START and END vertex are removed from the precedence graph, then one or more connected sets of vertices remains. Each connected set of vertices is a *parallel subgraph*. Estimates (or exact values) for the number of orders are found for each subgraph before the *parallel subgraphs* are recombined. This allows for the bounding values of the parallel subgraphs to be revised by adding in compensation values for each edge that is deleted or pair of vertices merged serially before the result of the parallel subgraph is combined with other parallel subgraphs.

Simplifying heuristics are applied to the graph at vertices with an in degree greater than one to reduce the graph complexity at these key points. Recall that vertices with an in degree greater than one are positions where either parallel merging or N type patterns could be used.

Algorithm: LOWER_BOUND(PARALLEL-SUBGRAPH-DATA)

```
1) Set CHECK-LIST to be empty (NIL).
2) For every remaining vertex I in the graph
      If NPARENTS[I] > 1 and MEET[I] <> NIL then add I to CHECK-LIST.
```



a) Near N shape : 11 orders possible

b) Modified to an N shape
8 orders possible

Figure 5-6: Lower Bound Transformations

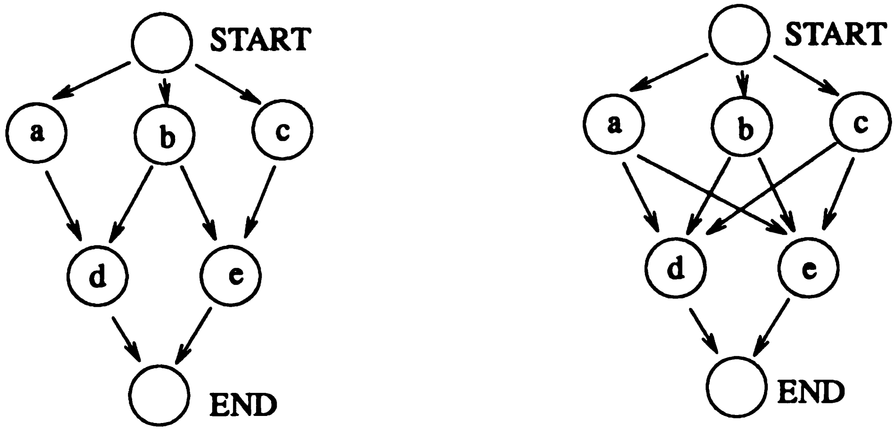
At this point all the vertices with more than one uncombined parent will be on the CHECK-N-LIST. We then apply the following heuristics to each vertex on the CHECK-N-LIST.

3) For each I on CHECK-LIST:

a) Reduce semi-series[17] vertices such as vertex 4 in Figure 5-6.

```
For J in PARENTS[I]
   If NPARENTS[J]=1 and NCHILDREN[J]=1 then
        Change graph so that
      PARENTS[J] = Add PARENTS[J] and PARENTS[I] lists - J index
      CHILDREN[J] = CHILDREN[I]
      N[J] = N[J] + N[I]
      O[J] = O[J] * O[I]
      If N_TEST(J) is TRUE then get the next I value and start over at Step a, otherwise proceed to step b.
```

b) Add constraints to serialize parallel segments. For example, given the precedence graph shown in Figure 5-7 a, edges are added to create the precedence graph shown in Figure 5-7 b. This allows the vertices {a, b, c} to be combined in parallel before being combined serially with the parallel combination of vertices {d, e}. By adding constraints, two disjoint sets of vertices are created such that all the vertices in one set must precede all the vertices in the other set. It is important to check and maintain ancestor lists for the graph so that a cycle is not added to the graph during this stage.



a) W shaped graph: 16 orders possible

b) W shape with added constraints
14 orders possible

Figure 5-7: Lower Bound Transformations

4) Try Series, Parallel & N pattern merge algorithms on the resulting simplified graph.
If the simplified graph can not be reduced to a single vertex then
Merge all remaining vertices using the series formula.

5) After the parallel subgraph has been evaluated, the number of orders stored at the remaining vertex is increased by one for each pair of vertices that was merged in a semi-series fashion, and one for each edge that was added to the graph.
Report the resulting lower bound stored at the only remaining vertex in the graph.

[17] All vertices in series have already been merged so additional constraints keep a series merge from being possible.

Step 5 is an attempt to at least partially compensate for the number of simplifications made to the graph. For each edge that was merged in a semi-series fashion a parallel merge with at least one other vertex was overlooked, cutting the number of possible orders by at least one. A closer bound could be obtained by taking into account a lower bound on the number of vertices that could have been merged in parallel. This would require the ancestor lists to be kept up to date each time a simplification was made to the graph so that a comparison of the the ancestors at vertex I and J could be performed. Similarly, the bound on the number of orders can be increased by at least one for each edge deleted from the graph. Adding an edge to the graph is only performed if the vertices were uncomparable before the edge was added. Thus adding the edge reduces the number of possible orders by at least one.

Developing useful heuristics to apply is an area for future research. Since we can have at most |V| vertices that are difficult to merge, it would be possible to spend O(|V|*|E|) time investigating different heuristics for each vertex without changing the overall order of the Oracle algorithm. For example, another heuristic that could be used in step 3 is to look locally around each CHECK-LIST vertex to see if changing a constraint would allow series, parallel, or N-type merges to further simplify the graph. Changing a constraint I → J → K to L → J → K or I → J → L (where L is a descendent of I and an ancestor of K) will produce a graph with a smaller number of possible orderings by restricting vertex J even more. Perhaps such a constraint movement would alleviate the merging problem and produce a better bound. After moving a constraint, it would be necessary to check for any redundant edges created by such a move. For example, in Figure 5-6 moving the constraint 1 → 4 to 3 → 4 and removing the redundant constraint 3 → END (which was created by moving the constraint 1 → 4) from the graph would create a parallel pattern which could then be evaluated.

The heuristics given in this section always underestimate the number of possible task orderings. The heuristic in step 3 a merges two vertices as if they were in series, when in fact additional parents of vertex I would have allowed for a larger number of possible orderings. Adding constraints as in step 3 b reduces the number of possible orderings by restricting some orders from occurring. After the heuristics are applied the series, parallel, and N type merging sequence is tried again. At this point any vertices remaining are reduced using the series formula which underestimates the number of possible orderings to insure that the resulting estimate is a lower bound on the number of orders.

The lower bound algorithm requires O(|E|) time to create the CHECK-list and reduce semi-series vertices (steps 1, 2, and 3a). Step 3b requires O(|V|*|E|) time to check the ancestor lists of each vertex being considered for additional constraints to make sure a cycle does not occur in the graph. Since step 4 calls up the series, parallel, and N-reduction algorithm it requires $O(|V|^2 * |E|)$ time and $O(|V|^2)$ space to perform. Thus, this lower bound estimation algorithm has an overall complexity of $O(|V|^2 * |E|)$.

5.4.2. Upper Bound

An upper bound on the number of possible task orderings can be obtained by applying heuristics to the graph resulting from the first application of the serial, parallel, and N pattern merging algorithms. The UPPER_BOUND() algorithm outlined below is used to find an upper bound on the number of possible task orderings. As in the case of the lower bounds a few graph simplifications are applied to vertices with an in-degree of two or more. By selectively deleting constraints and/or combining vertices in parallel we are guaranteed to produce a graph representing more task orderings than the original graph. If the resulting simplified graph can be reduced into a single vertex by applying the serial, parallel, and N pattern reductions a second time, then we report the resulting upper bound. Otherwise, all vertices remaining in the graph after the second reduction pass will be combined together using the parallel vertex formula to find an upper bound on the number of possible task orderings.

Algorithm: UPPER_BOUND(GRAPH-DATA)

1) Set CHECK-LIST to be empty (NIL).

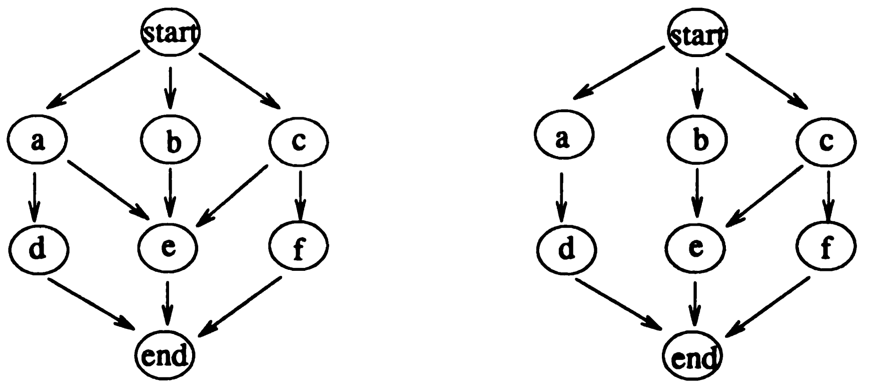
2) For every remaining vertex I in the graph

    If NPARENTS[I] > 1 and MEET[I] <> NIL then add I to CHECK-LIST

At this point all the vertices with more than one uncombined parent will be on the CHECK-LIST. We then apply the following heuristics to each vertex on the CHECK-LIST.

3) For each I on CHECK-LIST:

    This step selectively removes incoming constraints at vertex I in an attempt to make the graph an N-fold graph. It removes all but one of the incoming constraints that come from a parent vertex with more than one child. The algorithm tries to remove constraints that will increase the number of possible task orderings as little as possible. This is performed using a local graph topology check.



a) Non-N-fold Graph
66 possible orders

b) Modified Graph
74 possible orders

Figure 5-8: Upper Bound Constraint Removal Heuristic

For example, Figure 5-8 a shows a non-N-fold graph and Figure 5-8 b shows the heuristically altered graph that is an N-fold graph. All but one of the incoming edges to vertex e that are from vertices with two or more children are deleted.

Try the Series,Parallel & N pattern merge algorithms on this simplified graph.

4) If the simplified graph was not reduced to a single vertex then

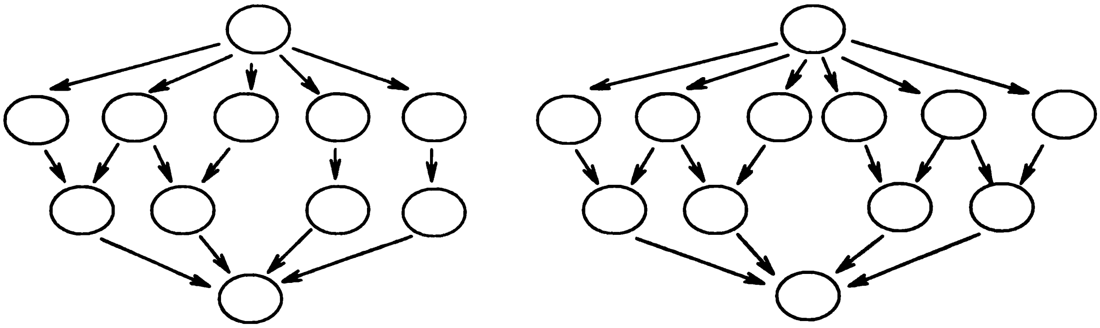
    Merge the remaining vertices using the parallel formula

5) For each edge deleted from the graph decrement the resulting bound on the number of possible orders by 1. Since only non-redundant edges are in the precedence graph, removing an edge makes the two vertices incomparable and increases the number of possible orders by at least one. This compensation could be more accurate if both the ancestor and descendent information for individual vertices is maintained in the graph. Deleting an edge can possibly create a much larger number of incomparable edges, this fact could be noted and the resulting increase in the number of orders could be compensated for in the final bounding value.

    Report the resulting upper bound stored at the only remaining vertex in the graph.

Note: Instead of deleting constraints as in step 3, problem constraints could be selectively loosened (analogous to the lower bound suggestion for tightening the constraints). Changing a constraint I → J → K to L → J → K (where L is an ancestor of I) or I → J → M (where M is a descendent of K) will produce a graph with a larger number of possible orderings by adding more valid positions for vertex J in the final graph. Any redundant edges created by moving constraints must be removed before evaluating the graph.

5.4.3. Lower and Upper Bound Estimates



a) lower bound < actual < upper bound
10,584 < 12,096 < 14,364

b) lower bound < actual < upper bound
49,392 < 64,512 < 90,972

Figure 5-9: Lower and Upper Bound Estimate Examples

Figures 5-6 through 5-9 give examples of the lower and upper bound heuristics and their effects on the number of possible orderings. The bounding values provide ball park numbers for how many orders are possible for the given graphs. When estimating heuristics are used for small parts of the precedence graph being evaluated these heuristics produce good bounds (for example Figure 5-9). Even when several estimates are necessary (such as in 5-9 b) the results can be good enough to decide roughly how many orders are possible. However, it is difficult to characterize the bounding values and to quantitatively predict their behavior on all graphs. This is due to the fact that the quality of the resulting bounding values depends highly on the local graph topology. Depending on the local graph topology a range from ball park to very tight bounding values can be obtained.

Ideally, extensions to the Oracle algorithm would allow even more graphs to be evaluated exactly, thereby reducing the need for using bounding values. (as discussed in Sections 4.7 and 5.6) However, future research into closer bounds and in rating the local performance of estimating heuristics would improve the closeness of the bounds. Using a branch and bound algorithm based on the *goodness* of the various local heuristics would allow the best of several heuristics to be applied for individual edges and to improve the final bounding values.

5.5. Overall Complexity of the Oracle Algorithm

Theorem 5.3: The Oracle has the following properties:

1) The exact number of task orderings is returned for N-fold graphs.

2) An upper and lower bound on the number of task orderings is returned for non-N-fold graphs.

3) The Oracle terminates for all possible input graphs in O(V^2 * E) time with an exact value or bounds.

4) The Oracle requires O(V^2) space.

The time complexity of the Oracle algorithm can be obtained by analyzing each of its disjoint parts. The complexity of the following stages can be thought of as being executed in series, so the largest time complexity of the parts is the time complexity of the Oracle algorithm.

1) Graph Initialization: The most time consuming initialization algorithm is for redundant edge removal. O(|V| * |E|) time is required to remove redundant edges.

2) Series & Parallel Reduction: O($|V|^2$ * |E|) time to check for parallel vertices.

3) N-Pattern Reduction: $O(|V|^3)$ time.

4) Upper and Lower Bound Heuristics: The number of orders for non-N-fold graphs is bounded above and below by two bounding values. This process requires $O(|V|^2 * |E|)$ time since after the relatively quick reduction heuristics are applied, the series, parallel, and N-pattern reduction algorithms are called again.

Each of the four categories listed above have time complexities of $O(|V|^2 * |E|)$ (or smaller). The space complexity for the algorithm is $O(|E|)$ to store the graph (remember that $|E| > |V|$). However, if bit-vectors are used for the comparison operations the space complexity becomes $O(|V|^2)$. If bit vectors are used and the original number of vertices is small ($|V| <$ word size of the computer) then the time complexity for the Oracle algorithm becomes $O(|V| * |E|)$. The next section discusses the experimental results obtained from using the Oracle algorithm.

5.6. Performance of the Oracle Implementation

The Oracle algorithm has been developed over the last 1.5 years. During the conceptual development of the Oracle algorithm several different Common Lisp implementations of the Oracle algorithm were coded on Sun 4 machines running Unix. Two significant versions of the Oracle algorithm were published [Mil90a, Mil90b]. The first version was capable of exactly determining the number of task orderings for the class of series parallel graphs (called simple fold graphs in [Mil90a]) and returning a single estimate for non-simple fold graphs. Appendix III compares the performance of the simple fold Oracle program with an enumerative approach for counting the number of different task orderings. The Oracle program significantly outperforms the enumerative approach. The Oracle program returns exact values or an estimate for the number of possible task orderings much more quickly than the enumerative program. In addition, the Oracle program has a low order polynomial run time, whereas the enumerative approach requires time proportional to the number of different task orderings (which can rise exponentially with the number of tasks).

The current version of the Oracle algorithm can handle N-fold graphs (which is a superset of simple fold graphs) and it also outperforms enumerative approaches as did the simple fold version of the Oracle algorithm discussed in Appendix III. The results presented in this section are for the current N-fold version of the Oracle algorithm which is implemented in Common Lisp. The performance of the N-fold Oracle program was obtained for a Sun 4/390 with 32MB of memory running under the Unix operating system. The Oracle algorithm is useful since it can determine the number of possible task orderings without enumeration for the class of N-fold graphs. Additionally, bounding values are provided for non-N-fold graphs. A class of labeled graphs with random edges was investigated to demonstrate the percentage of graphs that the Oracle algorithm could handle exactly without resorting to bounding values. As the number of vertices and number of edges are increased, the percentage of N-fold graphs falls off quickly. However, when either the number of vertices or number of edges is held fixed and the other varied, the percentage drops off quickly before rising back up again. In addition, plots of the cpu time required by the Oracle algorithm to produce exact values for N-fold graphs or to determine upper and lower bounds for non-N-fold graphs indicate that the Oracle algorithm has an $O(|V|^2)$ run time.

5.6.1. Correctness of the Oracle Program Code

Two major versions of the Oracle algorithm, an enumeration routine, and random graph testing routines have been implemented in Common Lisp and used on Sun 4 workstations running the Unix operating system. A total of 10,000 lines of Lisp code, comments and data make up these four different sets of software as described below.

1) Simple fold Oracle code: The first version of the Oracle algorithm consisted of 1200 lines of Lisp code and another 1300 lines of test data describing precedence graphs. The simple fold Oracle algorithm is described in [Mil89b], which also includes a

listing of the code and precedence graph data structures tested.

2) Enumeration code: A modified topological sort routine was created to enumerate precedence graphs in order to a) verify the correctness of the Oracle results, and b) compare the enumeration run times against the performance of the Oracle program as shown in Appendix III. It consists of 400 lines of code.

3) N-fold Oracle code: The latest version of the Oracle algorithm (described in this thesis) is implemented in 4000+ lines of Common Lisp code which includes display graphics and interactive editing code. The display graphics code uses *Common Windows* routines to interface to the X-11 windowing system running on Sun 4 workstations. The graphics code draws and modifies the precedence graphs as they are being entered by the user, or updates the graphics display as the precedence graph is evaluated by the N-fold or simple fold Oracle routines. The interactive editing code allows the user to enter, edit, and graphically modify a precedence graph using the three button mouse of the workstation.

4) Random graph testing code: The N-fold Oracle algorithm was tested on several million random graphs (as defined in the next section) to determine the experimental run time of the N-fold Oracle algorithm. The 2500 lines of Lisp code generate random acyclic graphs with a specified number of vertices and edges. After the random graphs are created they are evaluated by the N-fold Oracle routines. The internal Lisp run time required for the N-fold routine to find a solution is monitored and averaged over all the random graphs tested.

Verifying that all 10,000 lines of code do indeed produce the correct output for all possible inputs (according to the definition of each algorithm) would be a very difficult and time consuming task. Since, it was impractical to formally verify the code, many steps were taken to verify partial results and the final output from the different routines. Some of these verification procedures were:

1) The Lisp code was created in a hierarchical, structured format to aid in developing and testing individual routines to help insure that the routines performed as expected.

2) Numerous hand checks for the output of the different routines were performed.

3) For a large number of test cases, the enumeration routine results were compared to the results of both the simple and N-fold Oracle routines to make sure that all three of the programs agreed on the number of possible task orderings for simple fold precedence graphs.

4) Several internal program consistency checks were performed in both of the Oracle routines. For example, since both of the Oracle routines perform graph reductions, a count of the total number of vertices removed (reduced) from the graph is kept. If the total number of removed vertices ever exceeded $|V|-1$ ($|V|$ = No. of vertices) then a descriptive error message was printed and the program run terminated. These internal consistency checks were very helpful in debugging the initial program implementations of the algorithms.

5) The lower and upper bounds produced by the N-fold Oracle routines were checked by hand. In addition, when the N-fold Oracle produces a lower and upper bound the bounds are checked against each other to insure that LOWER_BOUND ≤ UPPER_BOUND. During the testing of several million random graphs this test always succeeded. Although this does not indicate that the lower and upper bounds did indeed bound the actual value of the number of possible task orderings for all the graphs tested, it indicates that the results are at least plausible.

6) Results were checked against published results from others.

The hierarchical and structured program development, numerous independent verifications of the program output, and internal consistency checks all point toward a correct implementation of the algorithms. Although these steps do not guarantee or prove the correctness of the algorithm implementations, they do contribute to the high confidence of the author that the program implementation for each of the algorithms is indeed correct.

5.6.2. Percentage of N-fold and Simple-fold Graphs

Unfortunately, no sufficiently large body of manufacturing tasks and precedence relations is available for testing the Oracle algorithm. Given the lack of sufficient real manufacturing test data it would be desirable to test the Oracle algorithm on all possible precedence graphs with, say N vertices and M edges. However, up to M = N*(N-1) unique directed edges can be added to a graph with N vertices. Thus, a total of N choose M = N! / (M! * (N - M)!) different graphs with N vertices and M directed edges can be created. For even fairly small values of N and M (say 10 or 15) the number of different graphs to be tested is prohibitively large.

In order to determine how effective the Oracle algorithm is in returning exact values for precedence graphs, randomly generated precedence graphs were investigated. Acyclic random labeled graphs with N vertices and M directed edges were investigated. Only acyclic random graphs are of interest[18], since no task orderings are possible in graphs with cycles. Four different data sets of graphs were investigated with the values for N = |vertices| and M = |edges| being varied or held constant as described below:

a) No. Vertices = No. Edges, both varied from 5 to 80.

b) No. Vertices = 2 * No. Edges, No. Edges varies from 3 to 40.

c) No. Vertices varied from 5 to 80, No. Edges = 15.

d) No. Edges varied from 5 to 80, No. Vertices = 15.

A random labeled graph with N vertices and M edges is constructed starting with N labeled vertices and no edges. Randomly selected edges are added one by one such that

[18] A topological sort can detect precedence graphs with cycles in O(|V| + |E|) time and space.

no duplicate edges are added to the graph and no cycle is created by the addition of an edge. The Oracle algorithm is applied to a large number of randomly generated acyclic graphs as described in the next two sections.

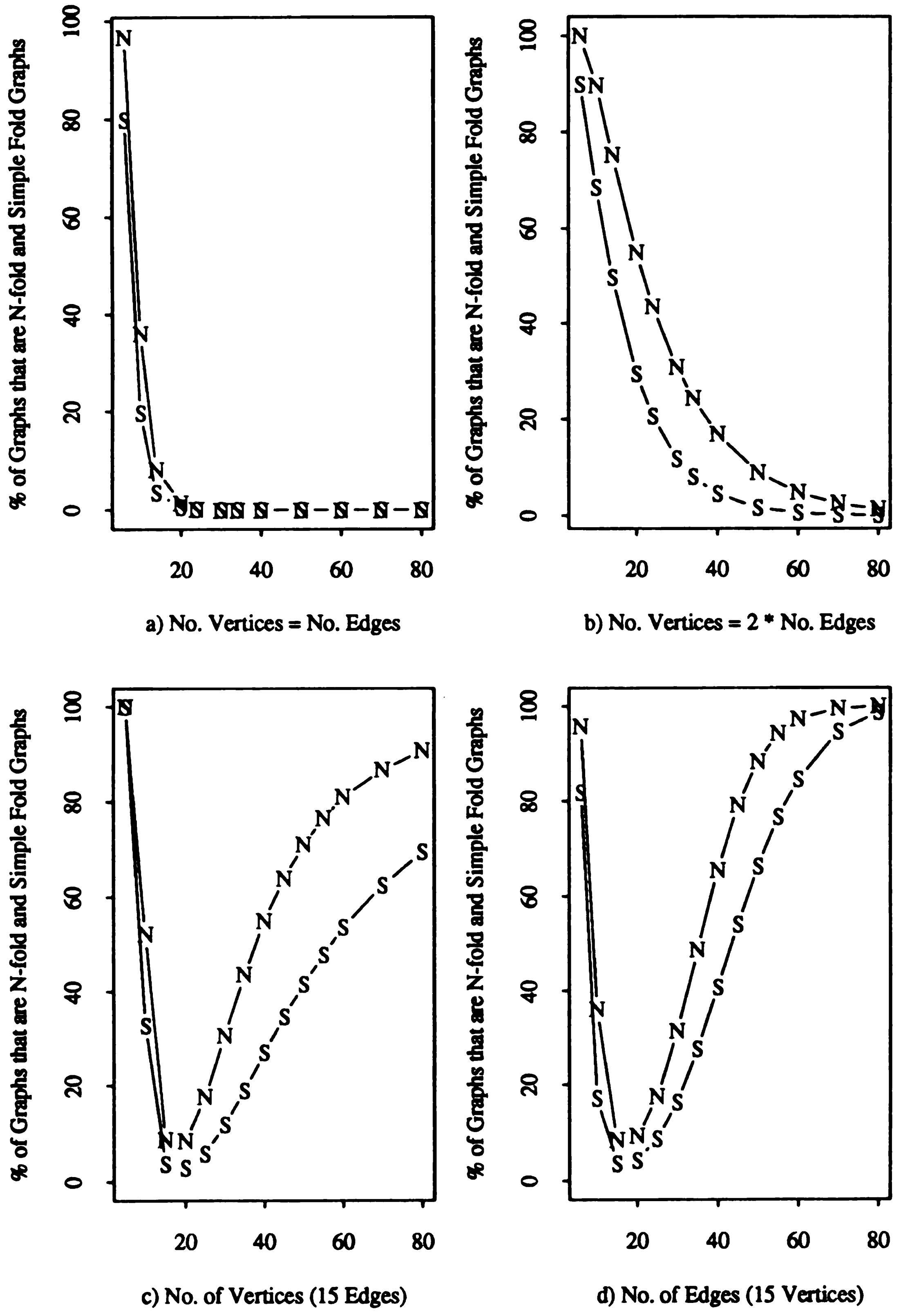
The percentage of randomly generated graphs that are N-fold graphs was obtained experimentally for a large number of random graphs from each of the four different classes of graphs. N-fold graphs can be evaluated exactly by the Oracle algorithm. Thus, if a large percentage of graphs in a class are N-fold graphs then the Oracle algorithm will produce exact results for most of the graphs in that class. An earlier version of the Oracle algorithm could only handle a subset of N-fold graphs called simple fold graphs [Mil90a]. To illustrate the improvement of the current Oracle algorithm over the previous one, the number of simple fold graphs was also counted. Additionally, the improvement suggests that adding a few more reduction patterns to the Oracle algorithm can further increase the percentage of graphs that can be handled exactly.

Figure 5-10 shows the percentage of N-fold graphs (N data points) and simple-fold graphs (S data points) for the four data sets. Figure 5-10 a shows that as both the number of vertices and edges are increased at the same rate, the number of graphs that can be evaluated exactly falls off sharply. This is due to the graph structure becoming more and more complicated, so that the series, parallel, and N patterns cannot reduce the whole graph. Figure 5-10 b shows that for less constrained graphs the percentage of N-fold graphs falls off more slowly. Additionly, the difference between the percentage of graphs that are N-fold and simple-fold graphs is readily apparent. These two graphs indicate that the ability of the Oracle algorithm to exactly evaluate graphs falls off when the number of vertices and edges becomes large. The use of subassemblies and scripts in process planning can help keep the number of different vertices in the graph to a reasonable number, thereby allowing the Oracle algorithm to exactly evaluate a large percentage of graphs without having to resort to bounding values.

The effects of varying either the number of edges or the number of vertices while keeping the other variable fixed is shown in Figures 5-10 c and d. For a fixed number of edges (or vertices) a large variation in the percentage of N-fold graphs can be found when different numbers of vertices (or edges) occur in the precedence graphs. Figure 5-10 c shows that the percentage of N-fold graphs varies as the number of vertices is increased in graphs with a fixed number of edges. The percentage of N-fold graphs starts out at 100 percent for six vertices and drops off quickly until it bottoms out around 18 vertices before rising back up again. A similar result holds for varying the number of edges while keeping the number of vertices fixed as shown in Figure 5-10 d.

When a small number of vertices and a large number of edges are placed in a graph (such as the first data point in data set c, or the tail end of data set d) the graph has many redundant edges. If unique edges are continually added to a small number of vertices (in an acyclic graph) a point is reached when no more edges can be added without causing a cycle in the precedence constraints to occur. A complete directed graph (a graph with a single directed edge between every pair of vertices) is called a tournament [Hara69]. If we further restrict the tournament to be acyclic, then only a single possible ordering of the vertices exists. After the redundant edges are removed from an acyclic tournament a single linear chain of vertices remains. When all the redundant edges are removed from graphs with a large number of edges and a small number of vertices, the non-redundant edge graphs are likely to be N-fold graphs -- or in the extreme case of acyclic tournaments -- represent a single possible ordering.

Graphs with a large number of vertices and only a few edges are likely to consist mostly of unconstrained vertices along with a few sets of vertices in subgraphs with a small number of constraints. Such graphs can often be reduced using the parallel, series, and N-fold reduction patterns since they are so simple and not cluttered with criss-crossing edges. This explains the gradual increase in the percentage of N-fold graphs for the last part of data set c and the quick falling off of the percentage of N-fold graphs for



a) No. Vertices = No. Edges

b) No. Vertices = 2 * No. Edges

c) No. of Vertices (15 Edges)

d) No. of Edges (15 Vertices)

Figure 5-10: % of N-fold and Simple-fold graphs in a Random Sample

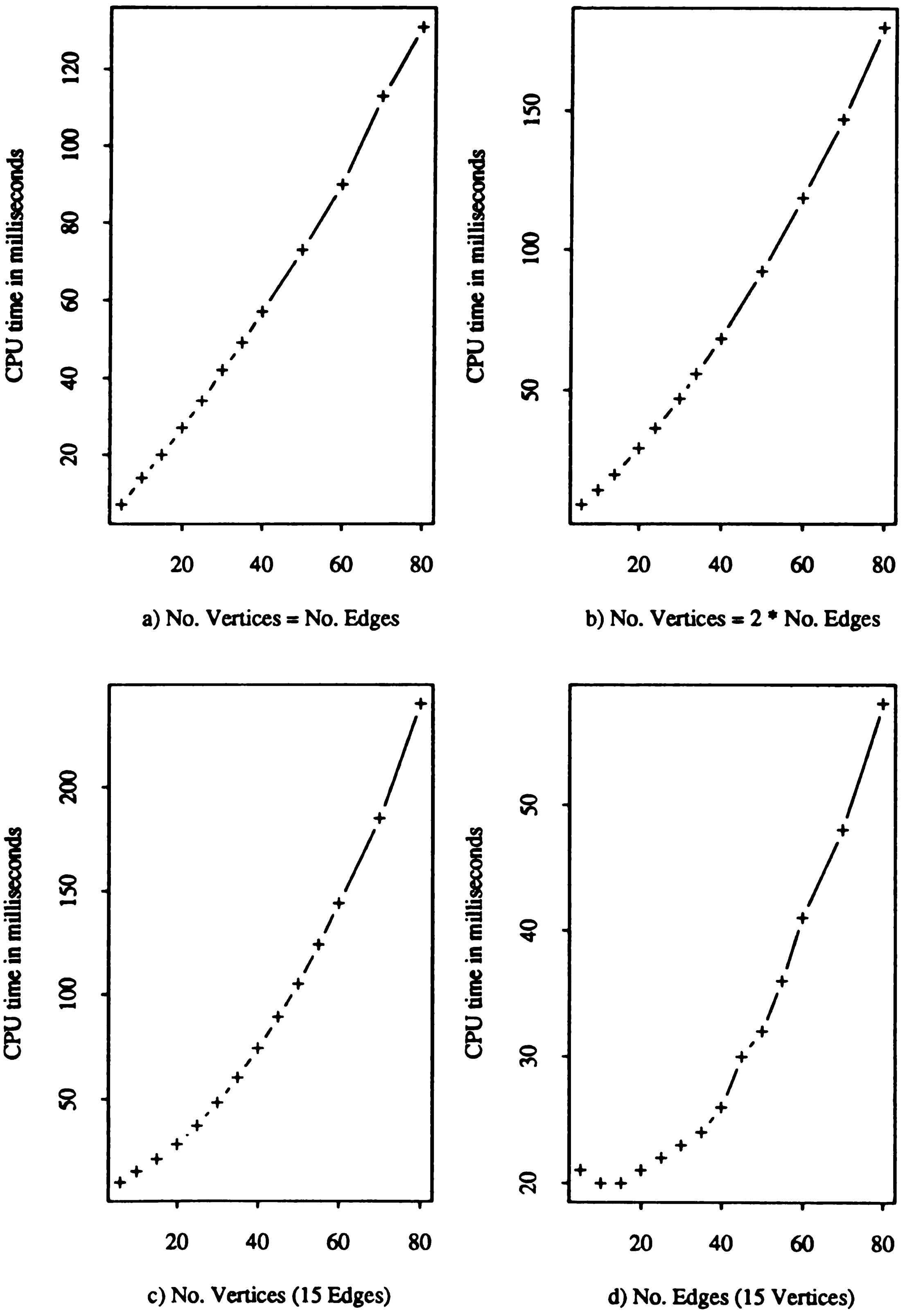
the first 5-15 edges in data set d.

5.6.3. Experimental Run Time for the Oracle

The Common Lisp implementation of the N-fold Oracle algorithm was timed on a Sun 4/390 with 32MB of memory (running Unix 4.0.3) for each class of random labeled acyclic graphs that was discussed in Section 5.6.2. The experimental run time for the Oracle algorithm to determine exact values for N-fold graphs or to determine that the graph is not an N-fold graph is shown in Figure 5-11 for each class of random labeled acyclic graphs (a decision problem). Each graph shows a run time of $O(|V|^2)$ time complexity. Portions of the graph show nearly linear relationships between the number of vertices (edges) and run times. In fact, least squares fitting shows that the run times appear to be closer to $\ln|V| * |V|$ time than $|V|^2$. This might be due to the lisp compiler optimization of the various searching operations used in the Oracle algorithm.

Figure 5-12 shows the run time for the Oracle algorithm to determine the exact values for N-fold graphs and return bounds for all non-N-fold graphs. The run times shown in Figure 5-11 did not include the time taken to return the upper and lower bounds for non-N-fold graphs (which is included in Figure 5-12). Figure 5-12 a shows that the Oracle only requires one half second on average to determine the number of different task orderings or bounds for 80 tasks and 80 vertices. Out of the 100,000 random graphs tested with 80 tasks and 80 vertices the maximum time required to return an exact value or bounds was 1.1 seconds. For such a large number of tasks and constraints the number of possible task orderings can be up to of $79! \approx 10^{117}$.

Obviously, trying to enumerate such a large number of different orders is computationally intractable. Whenever, a large number of tasks (i.e. more than 40) are to be performed, some method for reducing the number of tasks should be considered to avoid potentially enormous search spaces. For example, subassemblies, hierarchical planning, or subgoals could be used to cut down on the number of possible orderings. The Oracle



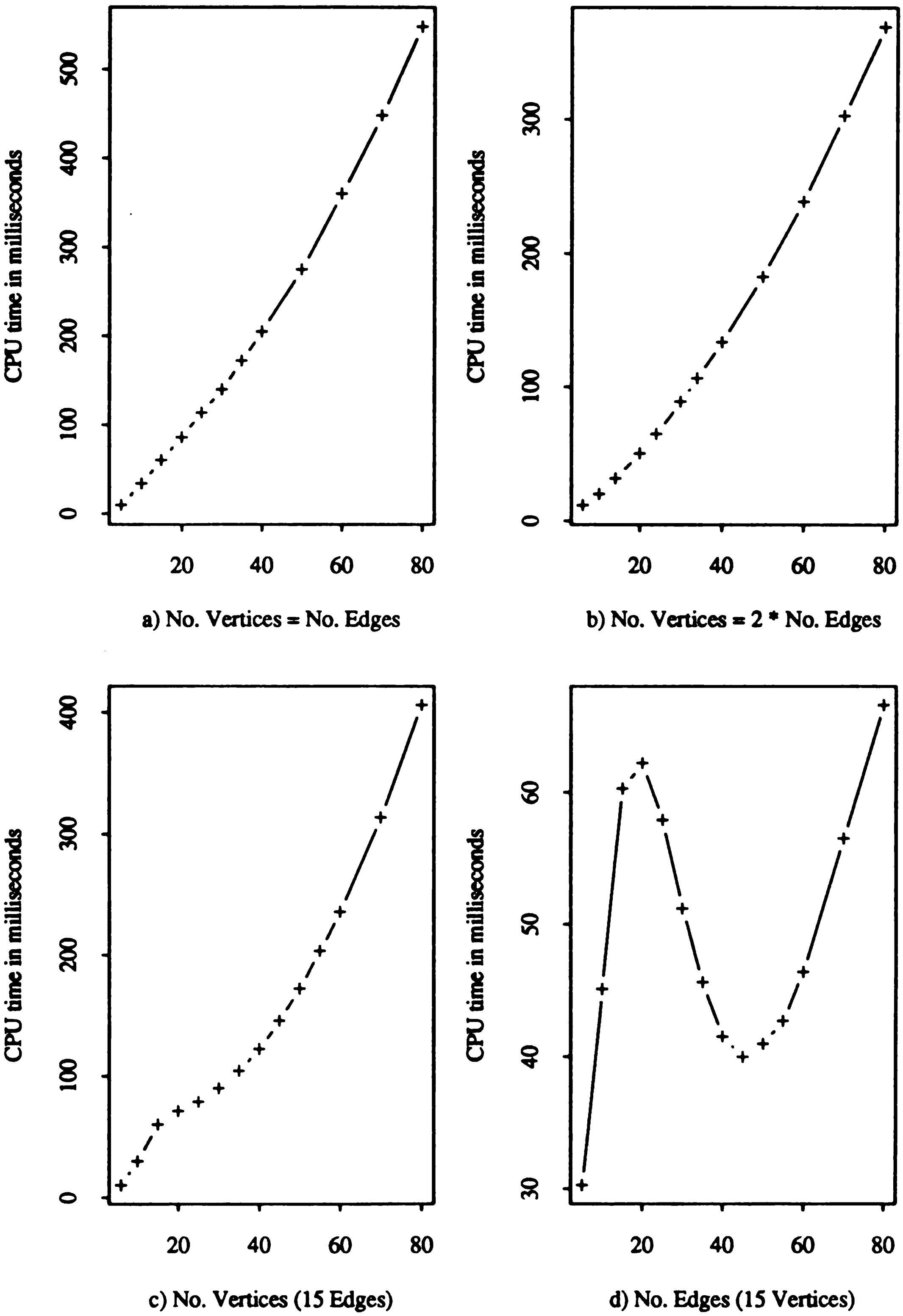
a) No. Vertices = No. Edges

b) No. Vertices = 2 * No. Edges

c) No. Vertices (15 Edges)

d) No. Edges (15 Vertices)

Figure 5-11: Oracle CPU Time for Random Graphs (Decision Problem)



a) No. Vertices = No. Edges

b) No. Vertices = 2 * No. Edges

c) No. Vertices (15 Edges)

d) No. Edges (15 Vertices)

Figure 5-12: Oracle CPU Time for Random Graphs (Exact Values and Bounds)

algorithm could be helpful when it is impossible to reduce the number of different tasks by allowing the user to keep adding constraints to the set of tasks until the number of possible plans becomes *manageable*.

Although the Oracle algorithm is a low order polynomial time algorithm, the time required to determine bounds for non-N-fold graphs is approximately three to five times as long as the time required to determine the exact number of orders for N-fold graphs. This explains the apparent deviation from the expected low order polynomial increases in run time as shown in Figures 5-12 c and d. In Figures 5-12 c and d either the number of edges or vertices remains fixed, while the other parameter is varied. The small *knee* in the CPU time plot of Figure 5-12 c and the much more pronounced *knee* shown in Figure 5-12 d is due to the large percentage of non-N-fold graphs in the knee areas relative to neighboring points in the plots. Since determining the bounds for non-N-fold graphs takes longer than determining exact values for N-fold graphs, points in the graph that have a larger percentage of non-N-fold graphs than neighboring points will require a disproportionately larger amount of CPU time.

5.7. Summary of Chapter 5

The graph representation, initialization, processing, and evaluation stages of the Oracle algorithm were outlined in detail. The three reduction formulas were stated and it was verified that they reduce N-fold precedence graphs while exactly evaluating the number of orders possible. Estimating heuristics were described that produce lower and upper bounds on the number of possible orders for acyclic precedence graphs. These heuristics were applied to several graphs demonstrating their use in evaluating non-N-fold graphs.

An overall $O(|V|^2 * |E|)$ time and $O(|V|^2)$ space complexity for the Oracle algorithm was derived, and supported by simulation runs. Several classes of random labeled acyclic graphs were used to experimentally determine the percentage of these graphs that are N-fold graphs which can be exactly evaluated by the Oracle algorithm. The Oracle algorithm is best at exactly handling graphs with a small or large number of edges relative to the number of vertices, since many of these graphs are N-fold graphs. The author conjectures that everyday manufacturing precedence graphs will be more amenable to the Oracle algorithm than random labeled acyclic graphs. This is based on the observation of assembly and manufacturing problems in which the precedence constraints are often of a local nature, typically with some large objects or key tasks dividing the the resulting precedence graph into parts. For example, in the AAPF system, parts with a *large* cross section are more likely to have constraints than parts with small cross sections when ray casting techniques are used along a particular removal direction. However, in random graphs every edge is equally likely -- creating less clustering of constraints around key vertices and reducing the chance of finding separate subgraphs to be evaluated independently. Comparing the improvement of the current Oracle algorithm to its predecessor that could only handle simple-fold graphs indicates that an even larger percentage of graphs might be evaluated in polynomial time using additional reduction patterns. The experimental run time for evaluating the number of possible task orderings was shown to be $O(|V|^2)$ for four classes of randomly generated labeled acyclic graphs.

Chapter 6

Concurrent Design and Evaluation

A CAPP system is proposed for Concurrent Design and Evaluation (CDE) that can aid the human designer in producing cost effective designs. The CDE system is intended for concurrent design and evaluation of products using feedback from interactive process and assembly planning modules. The CDE system is intended to capture all the salient features of concurrent process planning in a single package and can be developed using the technology available today. This chapter can serve as a high-level blueprint for an implementation of a design and evaluation CAPP system that interactively aides the human designer in producing designs that can be efficiently produced in a specific factory environment.

All CAPP systems are highly dependent on the process planning domain and factory environment. A CAPP system can only be built after a specific domain and manufacturing environment is identified. The manufacturing environment and product domain create a framework which grounds the process and assembly planning tasks. Concurrent design of the product with feedback from the manufacturing stage allows the resulting design to be optimized with regards to the plant facilities and standard machining procedures in order to produce a less costly product.

The CDE system is composed of the following independent data or program modules: a feature-based modeler and database for gathering information on the part being designed; a factory database including information on the machines, tools, and machining parameters; an assembly planner; a process planner; interactive assembly and process planning design critics; the Oracle algorithm; search and evaluation routines; and a robust simulation package.

A designer would interact with the feature-based modeler and the various CDE modules to produce a design that could be efficiently created in the flexible manufacturing environment which is modeled in the factory environment database. The current Oracle software package can be extended to be more effective in a CAPP package such as CDE. Several difficulties exist for the development of the CDE system and must be overcome before all the potential benefits of the proposed CDE system can be realized.

6.1. Process Planning Domain and Environment

The process planning function depends heavily on the manufacturing environment and domain [Chan85]. Since a general process planning system would have to handle all possible product types: discrete metal parts, plastics, wood, composites, sheet metal, etc., it is likely that the trend for creating domain-specific CAPP systems will continue. Obviously, the process planning task depends on the machines available in the factory that can create various machining features. A process planning system must have accurate knowledge of the machines and tools available on the shop floor along with the machining parameters and limits for the different processing and assembly functions in order to create efficient process plans. The nature and complexity of the process planning task requires appropriate knowledge representation techniques for the domain-dependent and factory-specific knowledge.

Although a CAPP system should store data useful for all the departments as described in Section 1.2, the description in this chapter is limited to the functions inside the dashed box shown in Figure 6-1. Figure 6-1 is adapted from the book by Nevins and Whitney on *Concurrent Design of Products and Processes* [Nevi89].

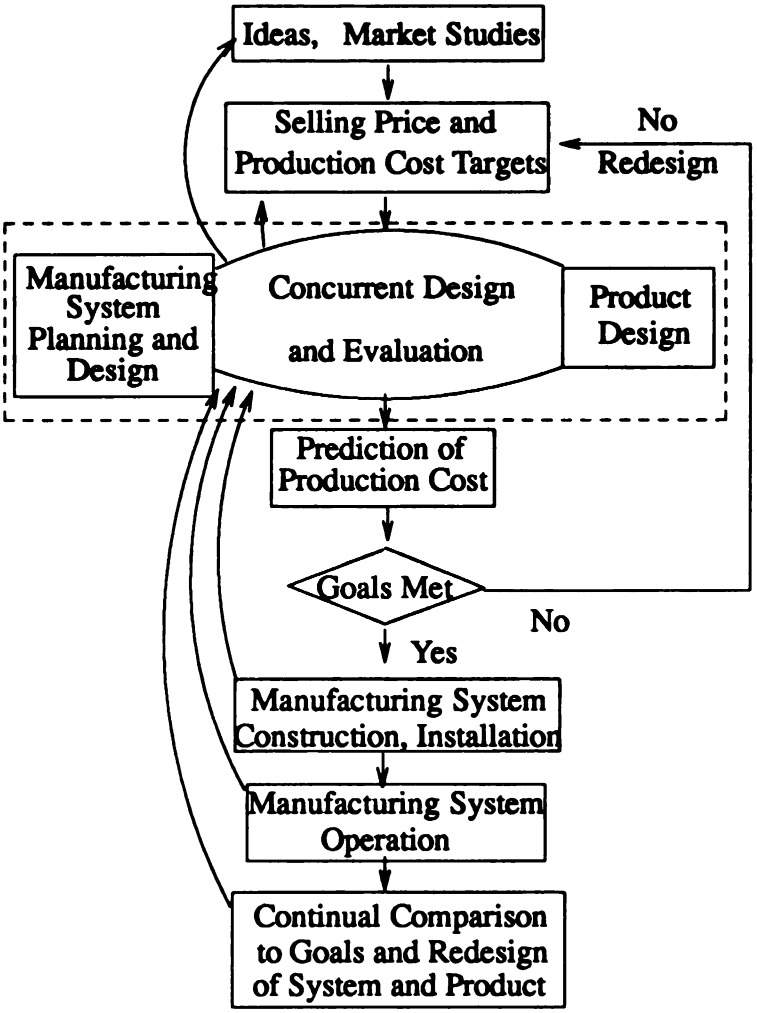

Ideas, Market Studies
Selling Price and Production Cost Targets
No Redesign
Manufacturing System Planning and Design
Concurrent Design and Evaluation
Product Design
Prediction of Production Cost
Goals Met
No
Yes
Manufacturing System Construction, Installation
Manufacturing System Operation
Continual Comparison to Goals and Redesign of System and Product

Figure 6-1: Integrated Design of Product and Manufacturing System [Nevi89]

6.1.1. Product Domain

An interesting domain for the CDE system would be discrete metal parts. A database of raw materials available for manufacturing the products is necessary to help determine the feasibility and cost of creating different product features and components. This database would include information on the various dimensions and material types of rod, bar, and other stock materials as well as the available fastening agents. This information would have to be readily available to other modules in the CDE system such as the interactive quick checking process planning module. For example, if the designer specified an axle diameter of 0.9" for a lawn mower and bar stock of 0.8" and 1.0" diameter were available, then the quick checking process planning module might suggest to the designer that an alternative diameter of stock be used to avoid unnecessary machining or processing steps.

The type and/or size of available fastening agents should also be kept in a global database. This would inhibit the designer from calling for non-existent screw lengths or trying to use fasteners that would not have the bonding strength necessary to hold the subparts together as called for in a design. Although the CDE system can warn the human designer about potential problems, the human designer should be able to override the warnings. For example, the designer may decide to order a new size of fastener or to create a new fastener in order to produce better designs. The interactive process planning module would try to help minimize the different types or sizes of fasteners used. Additionally, ease of assembly must be balanced with other fastener issues. The interactive process planner may suggest the use of rivets rather than screws to fasten parts together if the designer specified that the parts are unlikely to be disassembled during the life of a product.

6.1.2. Factory Environment

A minimal configuration of machine tool capabilities should include the four basic machining processes: turning, drilling, milling, and shaping, which can theoretically produce any contour on a workpiece [Groo83]. Ideally, a large array of operations would be represented in the manufacturing database to conform to a typical job shop. A larger set of machines and processes are [Groo80]:

machines: lathe, turret lathe, chucker, manual drill, NC drill, milling, shaper, planer.

processes: bore, cutoff, grind (surface, exterior or interior cylinder, centerless), broach, deburr, polish, buff, clean, paint, plate, assemble, inspect, package.

A modular construction of the factory environment database is desired so that entries can easily be added, modified, or deleted as new processes become available or conditions on the factory floor evolve. The capabilities, tools, costs, and other factors associated with each machine must be stored so that the CDE system can find the most appropriate machine to create the features present in a part.

Boothroyd investigates the performance and economics of six different assembly systems ranging from operator assembly lines without part feeders, to an automatic *Universal* assembly center [Boot82]. A flexible manufacturing environment for the CDE system could include assembly cells with a set of part magazines to hold the parts and fasteners that comprise the product and two three-degree-of-freedom robots for assembling the product on a work carrier. The work in progress could be moved among machining and assembly cells using Automated Guided Vehicles (AGVs). Special work carriers are useful in orienting and transporting the parts and subassemblies from station to station.

6.2. The Concurrent Design and Evaluation System

The CDE system is intended to aid the human designer in producing designs that can be manufactured efficiently in a specific flexible manufacturing environment. As with any interactive computer package, the user interface is of key importance. The system must be easy to use with the product design, evaluation, and planning functions integrated into a single package. A menu-driven, graphical, window-based environment is desired to allow all of the CDE modules to be easily accessed and used.

Figure 6-2 shows the main components or modules in the CDE system. Adjacent modules in the figure share data and commands depending on the state of the design and user requests. The three-dimensional feature-based modeler is at the heart of the CDE system. The product design features, tolerances, material type, intended use, and other desired performance attributes are kept in the three-dimensional feature-based modeler

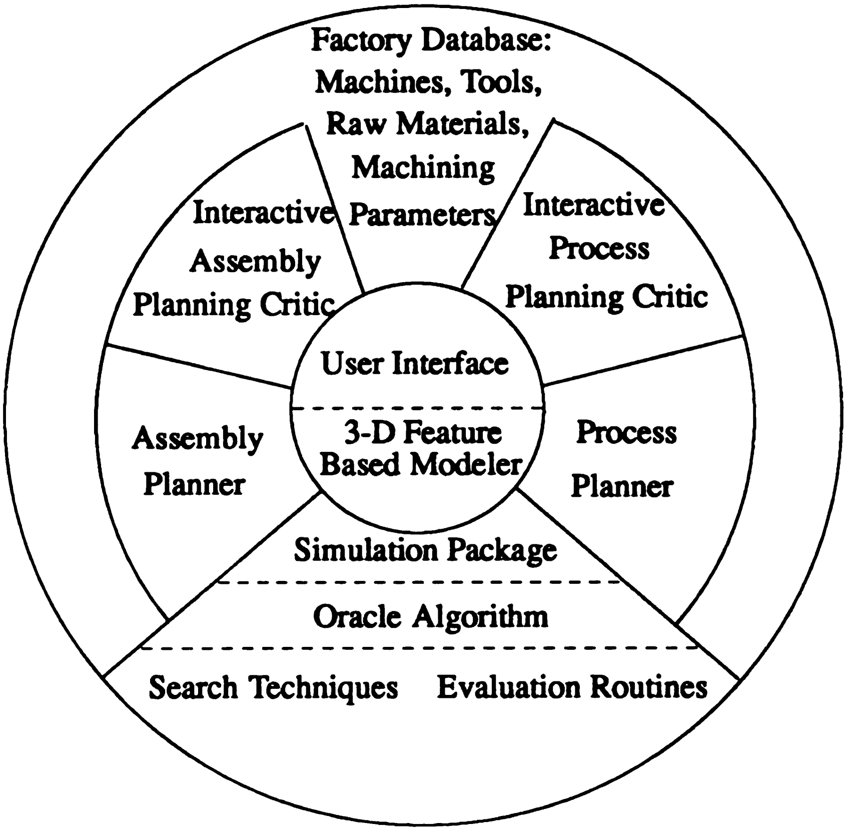


Figure 6-2: CDE System Interaction Diagram

database. Periodically, as the human designer is creating the product design, the interactive assembly and process planning modules check the current design for possible design flaws. Any potential problems found by the periodic design critics are highlighted in the feature-based modeler display window so that the human designer can decide whether or not modifications are necessary to improve the manufacturability of the product.

An additional function of the CDE system would be to categorize and save the part being designed for future reference. This could be performed using part classification codes or with the designer specifying the part type and function using predefined categories. If a part being designed is similar to a previously designed part, the CDE system can bring this fact to the attention of the designer so that a variant approach to the design and process planning tasks can be chosen.

A process and assembly plan can be developed using the assembly and process planning tools once the design has been entered. These tools use the factory database of machines, tools, and machining or assembly parameters to help determine an efficient process plan for the product. At the beginning of the design phase (and for various subplanning problems), the process or assembly planning modules can call up the Oracle algorithm when a fixed set of tasks is to be performed subject to precedence constraints. The tasks to be performed are derived from the features and geometry of the part by the assembly or process planning modules. However, the precedence constraints may be interactively entered by the designer and/or derived by the assembly and process planning modules. When the Oracle algorithm is invoked, a graphics window displays a graph or graphs with vertices representing the various tasks to be performed and directed edges between vertices representing precedence constraints. The designer can then add or delete constraints as desired in order to reduce the number of alternative possibilities or to increase the size or flexibility of a set of possible task orderings. Based on the number of possible task orderings for the various subplanning or planning problems, the system can recommend various search techniques to find an efficient plan. In addition to using different search techniques, the alternate plans can be evaluated at different levels of detail. For example, plans could be evaluated based on the actual robot arm motions and machine tool paths.

At any time during the planning stage the designer may request a graphical simulation of the machining or assembly steps for a particular plan. Previewing the machining or assembly steps can give the designer insights into ways to improve the manufacturability of the products, or it may highlight some invalid assumptions or constraints that had been placed on the planning problem. The interactive nature of the CDE system would allow the human designer to sift through a large search space of possible process plans in order to find a set of the most promising process plans. Each one of the *best* process plans could be evaluated in detail and ranked according to the predicted cost. This set of rated process plans could be stored along with the product design for use on the manufacturing floor. Additionally, each part and product would be coded for future reference so that it could be called up again if a similar part is designed at a later date.

6.2.1. Information Storage and Data Structures

An important issue in the development of the CDE system is to decide what information is necessary for the design, evaluation, and process planning tasks and how this information should be stored in the various CDE modules. Without actually developing the CDE system for a specific domain and factory environment, it is impossible to determine exactly what knowledge must be embodied in the CDE system and how this information should be stored. However, some general information categories and appropriate knowledge representation schemes for these different categories can be identified. The general information to be stored and used by each of the main CDE modules is:

Factory Database Module: Holds information on machines, tools, raw materials, machining paramenters, and processing techniques. Additionally, machine and tool geometry should be stored for use in detailed simulations. This compiled information must be kept up to date as factory conditions or manufacturing technology changes.

3-D Feature Based Modeler: Geometric product information is stored in CSG, B-rep, and other geometric formats. Part features and other attributes or characteristics can be stored in property lists (e.g. material type, part connectivity). Relations between parts in the product (such as wheels fitting loosely over axles) should also be stored as relational part attributes. All of this information is entered by the user or derived from user supplied information.

Interactive process planning critic: *Expert* process planning information is stored in rules which can be applied to the geometric and attribute data for the product being designed.

Interactive assembly planning critic: Assembly planning rules of thumb are stored in rules (similar to the interactive process planning critic).

Process Planner Module: Could store process planning knowledge in both procedural and rule based formats as appropriate.

Assembly Planner Module: Stores assembly planning expertise in procedural and rule based formats.

Oracle algorithm: Receives the set of tasks to be performed and the precedence constraints from the assembly or process planner modules that invoke the Oracle. The designer would be able to interactively change the set of tasks and/or constraints to aid in creating a desired number of different task orderings.

Simulation Package: Requests factory environment and machine geometry data from the Factory Database. Receives part geometry from Feature Based Modeler and machine/robot movement information from assembly or process planner modules. Simulation module may be used to act as a detailed evaluation routine.

Search Techniques and Evaluation Routines: Search techniques would require a set of tasks and constraints or some other appropriate search domain to be investigated. The search and evaluation routines may also require product geometry and attributes from the Feature Based Modeler. Factory environment, machining parameters, and costs can be obtained from the Factory Database as needed.

6.2.2. Factory and Domain Specific Database

All CAPP systems are specific to a particular domain of part/product types and include assumptions about the factory environment, machines available, and methods of creating various features. Section 6.1 described the importance of the factory and domain-specific environments and discussed the type of data that should be kept in the factory environment database. The machine, tools, machining parameters, and resources available for creating various parts must be made available to the assembly planning, process planning, evaluation and simulation packages in order to ground the CAPP system in the real-world. For example, plans can not be evaluated or rated without knowledge of the raw materials, machining processes, and tools available for manufacturing the final product design.

6.2.3. Solid Modeling Package

The solid modeling package is the heart of the CDE CAPP system. It must be capable of interfacing to other languages, expert systems, databases, and decision making programs. To facilitate the process planning task and to share the design data, a central and consistent method for representing part features, tolerances, material type, etc. must be available. Feature-based modelers are just now becoming available with more sophisticated systems under development. One way to make this data available to other modules would be to place the data in a standard data exchange format. Due to the expressiveness and information richness of the proposed Product Data Exchange Specification (PDES) standard, it would be the best choice for sharing CDE design data with other external display and processing modules. However, the PDES standard is under development, and only recently has the first draft of the specification been put forward. It is unlikely the PDES standard will be available and stable enough to be incorporated into the proposed CDE system in the near future. Thus, using the current Initial Graphic Exchange Specification (IGES) standard is the best remaining option for conveying geometric data to different reasoning or display modules outside of the CDE system.

As an alternate to using a standard such as IGES to share information, the geometric and attribute data could be requested by the planning, simulation, and evaluation modules from the feature-based modeler as needed. For example, if the solid modeling representation of the product being produced is readily available to the process and assembly planning modules, then very little if any information must be asked of the user. The solid modeling package should have an internal programmable interface that can be programmed to call and receive data or commands from programs outside of the solid modeler. For example, if the designer has added another part to the product being designed, the solid modeling program could call up the incremental process and/or assembly planning critics. These process and assembly planning critics would use expert system production rules to check for potential design problems that may cause unnecessary machining or assembly expense. This should enable problems to be caught early in the design stage.

Ideally, keeping the product design and specification information readily available in the solid modeling/product description database would keep out-of-date information from being used when the manufacturing engineer or marketing people reference the product data. One problem with relying on blueprints and other paper information is that old copies of the blueprints can be mistaken for the current blueprints. Additionally, storing and retrieving blueprints and other paper documentation can be difficult for large job shops or projects. For example, the sister cruise ships SS Constitution and SS Independence took some 80 tons of blueprints to describe [Kuth83]. Managing and keeping such a large amount of blueprints up to date is a difficult task.

In addition, having the geometric and part attribute information in a central location helps to free the designer from having to enter data twice. For example, having the features of the product readily available to the process planning module reduces the need to question the human designer about tolerance values for a feature if the user has already entered this data. If tolerance data is needed by the process planning module and it is not in the design data, then the needed data can be requested through an appropriate call to the feature-based modeler. The solid modeling window would be called up by such a request and the surface or feature that requires additional tolerance data would be highlighted. The response of the user would then be entered directly into the 3-D database for future reference.

6.2.4. Interactive Assembly and Process Planning Critics

Periodically during the design, creation, and editing stage, the interactive process and assembly planning modules would check the design for potential design errors or defects. For example, the interactive planner would highlight a planned hole if it were too close to the edge or if it would be difficult to machine to the desired tolerance. The user could request information on highlighted potential problems as desired. The interactive assembly and process planning design critics act as a *back seat* manufacturing planner that can be set to check the design at different intervals. These interactive feasibility critics attempt to eliminate some of the redesign cycles that occur when the designer and manufacturing engineer discuss a given design.

Potential problems the interactive process planning critic would check for:

1) Feature difficult to machine. Another surface may be restricting access or clearance above the surface to be machined.
2) Feature is too small or large to be made with current machines and tooling.
3) Part needs special machining and/or fixturing. For example, a thin sculptured part may be too delicate to withstand typical machining and fixturing forces.
4) Requested component is only slightly different from available raw materials. The designer may wish to change the component size to a readily available component or change the part dimensions according to raw material in stock.
5) Hole too close to the edge of a part (may break through) or hole has too high of a tolerance.

Examples of potential problems or inefficient designs that may be detected by the assembly planning critic:

1) *Overlapping* tolerances between pairs of mated parts. For example, calling for the insertion of a 1" diameter metal rod with an outer surface finish tolerance of ± 0.01" into a 1" ± 0.01" hole could result in rods that became *jammed* during insertion.

2) Lack of sufficient chamfering for a part to be inserted into a hole. Sufficient chamfering is particularly helpful when an automated assembly cell is being used.

3) Need for an intricate or large number of different fixtures. For example, highly sculptured objects or strange part shapes that do not have a good *base* part with planar surfaces for ease of positioning and fixturing are potential problems. Also, if many different assembly directions are necessary to add parts to a product during assembly, then the subassembly may have to be flipped or rotated periodically during assembly -- possibly even requiring different fixtures for each rotation.

4) The parts being assembled are too small or too large to be handled by the available assembly cells, (may require special tooling.)

5) Using a large number of different types and sizes of fasteners may require additional part magazines or increase the number of tooling changes necessary to assemble the product. Additionally, calling for fastener sizes or types not usually stocked would be flagged due to extra ordering and inventory costs.

6.2.5. Process and Assembly Planning modules

The process and assembly planning modules are invoked after the human designer has entered a tentative design. These modules use the geometric and feature-based information along with the attributes of the product design to determine efficient process and assembly plans for the current design. These modules would have to determine appropriate fixturing devices and setups, either interactively with the designer or automatically using a fixturing expert module [Boer88, Engl86]. A hierarchical approach to finding efficient plans is useful in decreasing the complexity of the assembly and process planning tasks. A total of five different command levels are described in [McLe83]: facility, shop, cell, work station, and equipment. At each level, knowledge about commonly used subsequences of commands could be applied to help limit the number of different subsequences of tasks that need to be checked. The concept of using *scripts* for subsequences was discussed in Section 2.4.4. For example, creating a high tolerance 1/2" hole could be accomplished using one of the following subsequences of operations: *(drill, ream)*, *(drill, hone)*, or *(core, ream)*. In order to help control the number of possible options available at the different levels, a combination of top down and bottom up planning may be helpful. The process and assembly planning modules should incorporate computer-aided inspection and testing steps into the final process plan to verify that key components and subassemblies are within the desired specifications.

The tasks to be performed and the precedence relations among the tasks can be derived by the assembly and process planning modules for some of the process planning subproblems. However, some of the precedence constraint knowledge is difficult to determine automatically and should be entered by the designer [Bour84, DeFa87, Huan89]. The assembly and process planning modules can call the Oracle algorithm to get an idea of the search space size whenever a set of tasks is to be performed subject to precedence constraints. Once the size of the search space is known, appropriate search and evaluation techniques can be chosen based on the number of possible orderings. For example, the high-level machining and assembly steps could be evaluated to determine approximate costs for moderately sized search spaces or in order to get rough cost estimates for a plan. However, large search spaces may require some type of heuristic search to cut down on the number of different task orderings to be evaluated. A large search space can be investigated to find the most promising alternatives, and then a detailed simulation of the most promising alternatives can be used to choose the most promising plan.

6.2.6. Oracle Module

The Oracle algorithm is called up by the assembly and process planning modules when a set of machining or assembly tasks has been decided upon and a set of precedence constraints restricts the possible task orderings. For example, given a primary assembly direction[19], precedence constraints on the assembly of the components in a product can be derived based on potential collisions between parts during the assembly. If the assembly of part A along the primary assembly direction would cause a collision with parts B and C if parts B and C were already assembled, then the precedence constraints $A < B$ and $A < C$ would be indicated. Any valid assembly sequence along the primary assembly direction must not violate any of these derived precedence constraints.

Other precedence constraints can be obtained from the designer as discussed by [Bour84, DeFa87, Huan89]. This process of gleaning the precedence constraints from the user could be facilitated by having the feature-based design and machining information on line in the CDE package. The designer could use the CDE system to explore alternatives in applying or removing precedence constraints to the assembly or process planning tasks and have the Oracle algorithm determine how many different task orderings were possible for each alternative. Depending on the intentions of the designers, they could increase the flexibility of the set of plans or reduce the number of possible sequences to restrict the search space so that it would be less costly to find the most efficient solutions.

6.2.7. Search / Evaluation / Simulation

A tool box of possible search and evaluation routines should be available in the CDE system. Some search techniques to consider are heuristic searches, A^*, branch and bound, beam search, and other alternative search methods for different search domains.

[19] One of Boothroyd's rules for *Product Design for Assembly* is to try and build the product in a layered fashion with each part being assembled from above [Boot82].

The heuristic search category would include a variety of different domain-dependent heuristics that prune the search space using simplifying assumptions -- which may result in a suboptimal answer. Although using an A^* search technique with an admissible evaluation function is guaranteed to find an optimal solution (with respect to the evaluation function), it may require a large amount of time and/or memory to conduct the search.

Different evaluation functions could be applied in evaluating partial or complete process plans. These evaluation functions could rate high-level plans or evaluate plans in detail down to actual robot arm motions and grasping positions for a specific simulated factory environment. Either the human designer or assembly and process planning modules would select the type of search and/or level of evaluation used in searching for efficient process plans. In addition to the evaluation functions, typical analysis of designs such as finite element analysis should be available in the CDE system to verify certain properties of the designs.

The simulation package in CDE should be able to represent the factory environment: machines, robot arms, fixturing, and the product being machined or assembled. Showing a simulation of a proposed assembly or process plan may give the designer ideas on how to improve the design and aid in the selection of alternative plans. Collision detection between objects in the environment and subassemblies or machines in motion is desired in order to validate proposed process plans. Verification of the validity and cost of different process plans can save time and effort by keeping infeasible or inefficient process plans from being attempted in the factory only to find out about the plan deficiencies at that time.

6.3. The Oracle Software

The current Oracle software has been an aid in the interactive investigation of the number of sequential orderings represented in precedence graphs. The experiments and results on the number of possible task orderings presented in Chapters 4 and 5 were obtained using this software package. The Oracle package is implemented in Common Lisp and uses the Common Windows graphical interface to access the X windows environment for entering and displaying precedence graphs. Additional code can be invoked to test the Oracle algorithm on sets of randomly generated precedence graphs.

6.3.1. Current Implementation of the Oracle

The Oracle software is an interactive, window-based, menu-driven graphical package for representing, displaying, and investigating the number of different sequential orders represented in a precedence graph. The Oracle software has been developed, extended, used, and tested for over a year. The code is well documented and modular so that various subroutines or functions can be easily modified or replaced. The package allows a user to quickly *draw* a precedence graph on the screen using the mouse and to investigate the number of different sequential orderings represented by the graph. Vertices and directed edges are added to or deleted from the graph using a mouse and several modifier keys. Menu choices exist for running the Oracle algorithm, enumerating the number of possible task orderings in the graph, changing the display scale (to accommodate larger graphs), and saving or recalling previously defined graphs. Several other choices allow for testing and verifying proposals and statements made by other researchers.

6.3.2. Extensions

The Oracle algorithm can be enhanced and extended to maximize its usefulness for the proposed CDE CAPP system. The proposed extensions are:

1) An additional routine should be added to translate from general precedence constraints to a set of precedence graphs that can be evaluated individually. Currently, this step is done by hand. Each one of the resulting precedence graphs could be

displayed in different parts of the graphics window (which is scrollable) or the graphs could be displayed one at a time as they are evaluated or investigated.

2) Adding several more reduction patterns to the Oracle to increase the set of graphs that can be handled exactly.

3) The graph identity shown in Figure 4-11 could be intelligently applied to graphs in order to exactly solve a larger class of graphs. Unfortunately, every application of this identity increases the number of precedence graphs to be investigated so it must be carefully applied.

4) Investigate the possibility of enumerating small non-*N-fold* subgraphs with less than eight vertices in order combine the result exactly with other N-fold subgraphs and produce better bounds and a larger percentage of exact answers.

5) The upper and lower bounding values could be improved as described in Section 5.4. This method would require keeping the ancestors and descendents of each vertex up to date as the graph is evaluated. Then, whenever a simplification heuristic is applied, the resulting upper or lower bounds can be made closer to the actual values.

6.4. CDE: Theoretical Research and Implementation Issues

Some aspects of implementing the CDE system will require additional research effort to overcome the current limitations and assumptions made in addressing various CAPP tasks. The CDE system could serve as a testbed for developing and comparing different methods for solving various subproblems of CAPP. This is one reason for the modular design of the proposed CDE system. As the system is being developed the different modules can be treated as black boxes with known inputs and desired outputs. Based on the assumptions and coding tasks finished at any given time, the other CDE modules can be developed and tested individually or with other partially or fully implemented components.

Some difficult theoretical issues that require additional research to create a robust CDE system are given below. In addition, references are given for each category to serve as pointers to some of the literature that partially addresses or discusses the difficulty of research in that area.

1) Ongoing research is progressing in the areas of feasible path and grip planning for robot manipulators. Current algorithms are very time consuming and do not produce optimal plans for complex environments [Dona87, Fers86, Shar86, Sand90].

2) Fixture design and selection of setups to constrain parts during the assembly and machining stages is difficult to automate [Boer88, Engl86].

3) Correctly determining the stability of arbitrarily shaped subassemblies is intractable in general, but some special cases can be handled [Palm87, Hof90a].

4) Finding optimal plans and schedules given a large number of alternative plans is difficult. This is especially true when it is computationally expensive to evaluate each of the possible alternatives accurately [Sand90, Fox85, Whit88]. Hoffman has recently developed reuse theorems that can be applied to deduce some freedoms of motion for new assembly states from previous calculations rather than computing everything again [Hof90b].

5) Reasoning about uncertainties in the assembly environment and determining robust recovery plans is a difficult problem [Hutc90].

6) Investigating how tolerances affect finding efficient and even feasible assembly plans is an area of ongoing research [Sand90].

Some parts of the proposed CDE system already exist and should not require additional research, but instead require a good deal of implementation effort. Software vendors are starting to market feature-based modelers that would work quite well in a system such as CDE. However, the problem of getting these commercial packages to interface with the other modules in CDE may be difficult.

Implementation is not always easy. Combining existing theory and knowledge to create a result that is greater than the sum of its parts can be difficult. Although the U.S. has been known for producing new ideas and good research, it is not as well known as other countries at creating efficient implementations which take advantage of the time and cost savings that are possible using existing technology and methods. In a competitive industrial environment creativity is essential, but it is equally important (if not more so) to put ideas into action in order to gain a competitive advantage.

Many search techniques are known and only need to be implemented for the CDE framework. However, determining which techniques are best suited for the different types of search spaces arising in the CDE system is an area for further investigation. Although some evaluation techniques have been discussed by various researchers, the development of accurate evaluation functions may be difficult. Various packages exist for simulating motions in a 3-D environment but are not without assumptions and limitations. If an appropriate simulation package for the CDE system does not come bundled with a feature-based modeler, then additional work may be required to create a useful system for simulating the machining and assembly environments.

Chapter 7

Conclusion, Discussion and Areas for Future Research

The future for computer-aided process planning seems bright. Good progress has been made in many areas such as restricted part type CAPP systems, knowledge engineering, geometric solid modelers and the current development of feature-based modelers. As the ranks of expert human process planners continue to decline, the need for CAPP systems will grow. Much more research needs to be done in trying to build more general CAPP systems and integrate CAPP into the CAD/CAM environment, as discussed in Chapter 6, so that the transition from the design of the product to its manufacture is smoother and more free from unnecessary human intervention. A good system would also be able to explain how the final process plan was constructed. Knowledge engineering plays a key role in this development.

7.1. Summary of Thesis

The survey of variant, generative, and artificial-intelligence-based process planning systems showed that there are many difficult representation and processing issues involved in CAPP. This thesis has investigated the varied costs associated with different process plans and the number of feasible process plans. There are many similar combinatoric and geometric reasoning tasks that occur in both the manufacturing and assembly planning domains of CAPP. Some difficult processing problems were investigated in the development of an Automated Assembly Planning with Fasteners (AAPF) system that reasons about fasteners and the assembly planning task.

The AAPF system works directly from the CAD model of the product to be assembled [Mil89a]. In order to avoid a combinatoric explosion of possible assembly sequences, the AAPF system assumes that no internal or external forces occur during the assembly process, and therefore, the reverse of a disassembly sequence is a valid assembly sequence. Starting with an assembled product, finding a disassembly sequence is much easier than finding an assembly sequence because the number of feasible removal directions for each of the parts in a product being disassembled is limited by the other parts in the assembly. Additionally, if the disassembly can be accomplished with each step consisting of the complete removal of a part from the assembly (monotonic assembly), then no dead end states can be reached that require backtracking. However, dead end states can occur if an assembly planner tries to find a valid assembly sequence starting with all the parts disassembled.

To the best of our knowledge, the AAPF system was the first system to derive fastener and component connectivity from geometric information to determine which components were held together by which fasteners. Additionally, it was the first system to reason about individual fasteners and to use heuristics based on fastener type, size, and location in an attempt to derive efficient disassembly plans. Although these heuristics were designed to produce efficient assembly plans, the heuristics did not model fixturing constraints. When the AAPF plans were evaluated according to both fastener and fixturing constraints, the results were mixed: some plans were efficient and others required more time than hand-generated plans that accounted for fixturing. At least two different process plans were evaluated at a high-level for each of several different objects. The evaluations show that inefficient plans can require up to 100% more time than efficient plans to produce a product in a flexible manufacturing environment. The large variance in the time required to manufacture and assemble a product under different plans indicates the importance of searching for an efficient plan. Unfortunately, many feasible process plans may exist, making the selection of an efficient plan a difficult task.

Different heuristics, search techniques, and evaluation functions can be used to help overcome the problem of a large number of possible solutions. For example, if we know that many possible task orderings exist, then heuristics can be used to prune the search down to the most promising sequences. Less accurate heuristics may even be necessary in cases when an extremely large number of feasible orders exists. However, if we know that a small number of orderings exist, then each solution can be evaluated in detail to find the most efficient sequences. Using heuristics on small search spaces risks pruning an *optimal* sequence from further consideration. Additionally, varying levels of detail can be used to evaluate alternative plans depending on the number of different orders possible.

The number of possible task orderings can also be used to select the most flexible set of assembly or machining plans from alternate sets of plans that use different fixturing devices or approach directions. The set of plans or operations with the largest number of possible sequential orderings would tend to have more flexibility than a smaller set of feasible task orderings. For example, having a large number of alternative assembly sequences available is beneficial when individual components arrive at random times because fewer buffering operations are needed [Home90].

The general problem of determining the number of possible task orderings for a set of tasks and precedence constraints has been shown to be #P-complete (sharp-P-complete) [Wink90]. This indicates that the problem is intractable today and will remain intractable until someone can show that the NP-complete class of problems can be mapped to the class of P problems in polynomial time (considered very unlikely). Thus, the approach taken in this thesis was to find a class of planning problems that can be evaluated exactly in polynomial time and to calculate upper and lower bounds for all other problems in polynomial time. To facilitate determining the number of possible task orderings, a precedence graph representation G=(V,E) is used. Tasks to be performed are represented by vertices (V) in the graph, with precedence constraints being represented as directed edges (E) between vertices.

An *Oracle* algorithm was described in Chapter 4 that can determine the number of task orderings for a class of *N-fold* graphs and produce upper and lower bounds for non-*N-fold* graphs. The Oracle algorithm was shown to have $O(|V|^2 * E)$ time and $O(|V|^2)$ space complexity. Simulation results show that the Oracle exhibits an $O(|V|^2)$ run time behavior for several classes of random graphs. The Oracle uses three unique subgraph patterns to successively merge vertices in the precedence graph representation while keeping track of the number of possible orderings as each subgraph pattern is applied. The upper and lower bound calculations take advantage of several heuristics that are guaranteed to produce results that overestimate or underestimate the number of possible task orders.

A CAPP system for Concurrent Design and Evaluation (CDE) of products is proposed but not implemented, which would use the Oracle algorithm to aid the designer and manufacturing engineer in finding an efficient process plan. The main modules and interactions between these modules is investigated to show how various representation and processing issues could be handled. An example is given to show how the system would be useful in guiding the designer to produce a product design that can be manufactured and assembled efficiently in a flexible manufacturing environment.

7.2. Contributions of the Thesis

The main contributions of this thesis are:

- Investigating the importance of modeling and reasoning about fasteners in the assembly planning domain through the development of the Automated Assembly Planning with Fasteners (AAPF) system.
- Emphasizing the large variances possible in manufacturing and assembly times for a product depending on the process plan developed.

- Demonstrating that a large number of feasible process plans may exist, even for simple products, and that it is desirable to know how many feasible plans exist to help in finding an efficient plan.
- Developing the graph representation for a set of constrained tasks and transformations which reduce complex graphs into simpler graphs.
- Creating and analyzing an Oracle algorithm that quickly determines the number of possible task orderings for a set of tasks and precedence constraints. Exact values are returned for *N-fold* graphs and upper and lower bounds on the number of orderings are returned for non-*N-fold* graphs.
- Describing a practical CAPP system for concurrent design and evaluation of products which uses information on the number of possible task orderings to aid the human designer and manufacturing engineer in producing efficient process plans.

7.3. Discussion and Areas for Future Research

The Automated Assembly Planning with Fasteners system did not account for fixturing and stability constraints. Fixturing constraints and the stability of subassemblies are difficult to address in CAPP, but progress is continually being made. Recent CAPP systems or assembly planners addressing fixturing problems are: [Boer88, Wolt89, Huan90].

Determining whether or not components in an *assembled* configuration will be stable[20] is a very difficult problem to solve. An *assembled* configuration can be an actual product assembly or a collection of objects positioned in three-dimensions (perhaps stacked upon or leaning against other objects). Palmer has shown that even in two-dimensions, determining whether or not a given configuration of polygons is stable under

[20] For a collection of objects to be considered stable, the objects must be able to stay in place (i.e., without falling to the ground due to gravity) even when a small force is exerted on some of the objects. An egg balanced on one of its ends is not considered to be stable.

certain conditions is generally intractable [Palm87]. Since a collection of polygons in two-dimensions can be extruded to form three-dimensional objects, this result indicates that under certain conditions it will be intractable to determine the stability of 3-D objects. In the realm of CAPP systems, Hoffman has implemented an instability test in his assembly planning system that is sufficient but not necessary for detecting instability [Hof90a].

Difficult problems remain for CAPP in the areas of motion planning and collision detection. A valid assembly sequence requires additional information on valid grip points, stability and fixturing issues, part tolerances, and path planning. Optimal path planning in three-dimensions, even among polyhedral objects, is itself a problem which is known to require significant resources [Shar86]. Automatically determining disassembly sequences is difficult when arbitrary translation directions and rotations are allowed due to the infinite number of choices for the direction and rotation of components during the assembly process. Additional problems occur in determining how far to move a component in any one direction before changing directions when multiple sequences of translations and rotations are necessary in creating an assembly plan. Removal direction and distance heuristics such as those implemented by Hoffman are able to overcome some of these difficulties in automated assembly planning for complex objects [Hof90a].

The Oracle algorithm exactly determines the number of possible task ordering for *N-fold* graphs and returns upper and lower bounds for all other graphs. Several methods are available for increasing the number of graphs that can be handled exactly. The most obvious choice is to investigate using other reduction patterns in addition to the series, parallel, and N shaped reduction patterns. Additional patterns that may be useful include a W shaped pattern and an N pattern with a vertex along the diagonal *edge* of the N. These new patterns must be studied to determine how they would affect the overall time complexity of the Oracle algorithm and the percentage of graphs it could exactly handle.

Figure 4-11 shows a graph identity that can also be used to increase the set of graphs that can be evaluated exactly by removing edges that are difficult to evaluate. This is accomplished by creating two subproblems that can be solved independently. In a parallel environment each of these subproblems could be given to different processors and the results combined when each processor finishes. Another method to increase the number of graphs that can be handled exactly is to enumerate small subgraphs of vertices that are non-N-fold graphs. If the subgraphs are of size $|V| \leq 7$, then at most 5040 orderings would have to be evaluated. The resulting values for these subgraphs could then be combined with the remaining vertices using the reduction patterns.

As mentioned in Section 5.4, the upper and lower bounds can be made closer if both the ancestors and descendents of each vertex are kept up to date during the graph reduction process. This would enable a more accurate count of the number of possible orders that are being lost or added depending on the graph simplification heuristic used. These tighter estimates would require additional code but would not increase the overall complexity of the Oracle algorithm.

The Oracle was tested on a class of random graphs due to the lack of a large database of manufacturing and/or assembly tasks and constraints. This leads the author to wonder, "What is the nature of typical manufacturing tasks and constraints ?" Is it likely that real manufacturing graphs may have a higher percentage of N-fold graphs than random graphs with the same number of vertices and edges ? Could it be that 60% of typical automobile products correspond to N-fold graphs, but only 40% of graphs occurring in aerospace manufacturing tasks are N-fold graphs ? A study of the topology of manufacturing precedence graphs may suggest a set of additional reduction patterns which would increase the number of graphs that could be handled exactly by the Oracle algorithm. Additionally, such a study may indicate some fundamental properties of manufacturing or assembly tasks that may be helpful in identifying potential subassemblies in a product.

The field of CAPP should blossom in four to six years as engineering students graduate into the work force with a good grasp of how to use and apply computers and knowledge engineering to process planning and manufacturing. The availability of more versatile modeling and representation schemes would allow CAPP systems to go from raw product design and specification data to a complete and efficient process plan. Solution of the process planning problem for large search spaces, especially when efficient or 'optimal' plans are desired, requires careful thought and application of an appropriate search technique and representation. Chapter 6 describes the major components of the proposed CDE CAPP system for concurrent design and evaluation of products. The CDE system should be helpful in finding efficient process plans when many different plans exist. Although some of the components for the CDE system exist, other components need to be researched, developed, and enhanced to create a versatile CAPP system.

Appendices

Appendix I: Sample Input and Output from the AAPF System

Partial Input Data File for the Mouse Model

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: USER; Base: 10 -*-
;;; File: SILVER:>joe>oaf>data-mouse.lisp      Joseph M. Miller      Aug. 23, 1988

; List #1 Optional comment lists
;     1st sub list is the object name
((Partial Symbolics Mouse Definition)
;     2nd sub-list contains the names for each of the parts in the object
 ("black plastic case" "center mouse button" "right roller ball" "right roller holder")
;     3rd sub-list contains names for each of the fasteners
 ("right case screw" "right roller screw")
) ; end of the comment list

;  List #2 Set of objects (plus and minus primitives of type sphere,
; cone, cylinder, and block are possible) that make up the mouse assembly
;primitive + parameters                rotation        translationspositive/
;                          about x,y & z  along x,y, & z   negative type
;------------------------------------------------------------------------
;       Outer black mouse case and 3 mouse button holes defined
(((box 2.8 0.1 1.7        0.0 0.0 0.0    0.0 -1.15 0.0    p) ; Top of outer black case
   (box 0.8 0.1 0.3       0.0 0.0 0.0   -0.7 -1.15 0.0    n)  ; center mouse button hole
   (box 0.8 0.1 0.3       0.0 0.0 0.0   -0.7 -1.15 -0.55 n)  ; closest mouse button hole
   (box 0.8 0.1 0.3       0.0 0.0 0.0   -0.7  -1.15  0.55 n)  ; farthest mouse button hole
   (box 0.1 1.2 1.7       0.0 0.0 0.0   -1.35 -0.6   0.0  p)  ; left short side of box
   (box 0.1 0.2 0.2       0.0 0.0 0.0   -1.35 -0.1   0.0  n)  ; hole for mouse cord
   (box 0.1 1.2 1.7       0.0 0.0 0.0   1.35 -0.6   0.0   p)  ; right short side of box
   (box 2.8 1.2 0.1       0.0 0.0 0.0   0.0  -0.6   0.9   p)  ; farther long side of box
   (box 2.8 1.2 0.1       0.0 0.0 0.0   0.0  -0.6  -0.9   p)  ; closer long side of box
   (cylinder 0.4 0.1 0   90.0 0.0 0.0   -1.17 -0.9  -0.3  p)  ; leftmost screw boss
   (cylinder 0.4 0.1 0   90.0 0.0 0.0   0.71 -0.9   0.26  p)  ; rightmost screw boss
   )
  ((box 0.75 0.2 0.25     0.0 0.0 0.0   -0.7  -1.18  0.0  p)  ; center mouse button
   (box 0.75 0.05 0.40    0.0 0.0 0.0   -0.7  -1.07  0.0  p)
   (cone 0.2 0.1  0.05   90.0 0.0 0.0   -0.65 -0.99  0.0  p)
   )
  ((sphere  0.17  0 0     0.0 0.0 0.0   -0.8  -0.145  0.5 p)   ; right roller ball
   )
  ((cone  0.5  0.1  0.25 90.0 0.0 0.0   -0.8  -0.35   0.5  p)   ; right roller pillar
   (cone  0.22 0.15 0.20                90.0 0.0 0.0      -0.8  -0.21   0.5n)
   )
 )
;   List #3  Contains the FASTENER information.
;primitive + parameters                                   rotation        translations      positive/
;                                                    about x,y & z  along x,y, & z    negative type
(
 ((screw (0.06 0.01 0.016 0.018 0.001 0.9 0.8) 0 0  90.0 0.0 0.0  0.71 -0.45  0.26        p))
 ((screw (0.04 0.03 0.008 0.01 0.001 0.3 0.2) 0 0 270.0 0.0 0.0  -0.8  -0.55  0.5         p))
 )
```

Disassembly Sequence for the Mouse Model

```
((START-OF-DISASSEMBLY)
 (REMOVING SCREW FASTENER (right case screw) ACCESS ((X 1) (X -1) (Y 1) (Z 1) (Z -1)))
 (REMOVING SCREW FASTENER (left case screw) ACCESS ((X 1) (X -1) (Y 1) (Z 1) (Z -1)))
 (REMOVING (Black plastic case) FROM ((Y -1)))
 (REMOVING (center mouse button) FROM ((X 1) (X -1) (Y -1)))
 (REMOVING (left mouse button) FROM ((X 1) (X -1) (Y -1) (Z -1)))
 (REMOVING (right mouse button) FROM ((X 1) (X -1) (Y -1) (Z 1) (Z -1)))
 (REMOVING (base) FROM ((Y 1)))
 (REMOVING (left roller ball) FROM ((Y 1)))
 (REMOVING (right roller ball) FROM ((Y 1)))
 (REMOVING (main tracking ball) FROM ((Y 1)))
 (REMOVING (circuit board support) FROM ((Y 1) (Z 1) (Z -1)))
 (REMOVING (mouse cable) FROM ((X -1) (Y 1) (Y -1) (Z 1) (Z -1)))
 (REMOVING SCREW FASTENER (left roller screw) ACCESS ((X 1) (X -1) (Z 1) (Z -1)))
 (REMOVING (left roller holder) FROM ((X -1) (Y 1) (Z -1)))
 (REMOVING SCREW FASTENER (right roller screw) ACCESS ((X 1) (X -1) (Z 1) (Z -1)))
 (REMOVING (right roller holder) FROM ((X -1) (Y 1) (Z 1) (Z -1)))
 (REMOVING SCREW FASTENER (main roller screw) ACCESS ((X 1) (X -1) (Z 1) (Z -1)))
 (REMOVING (tracking ball housing) FROM ((X 1) (X -1) (Y 1) (Z 1) (Z -1)))
 (REMOVING FINAL PART (circuit board with mouse switches) FROM ((X -1) (X 1) (Y -1) (Y 1) (Z -1)
(Z 1)))
 (DISASSEMBLY COMPLETE))
```

*** Note: Possible disassembly directions are given for each removal operation above. e.g., The list '((X -1) (Y 1)) indicates that the part being removed can be removed along the negative X direction, or the positive Y direction without the component(s) being removed interfering with any of the components remaining in the object.

Appendix II: Evaluation of Process Plans

Each Plan in this appendix is evaluated at a high level for two different robot arm speeds in the factory environment discussed in Section 3.4.1. A summary of the results presented here appears in Section 3.4.2. In practice, plans that scored highly according to this high level evaluation function would be evaluated in more detail to determine more precisely the projected time cost of each plan. The high level evaluation function rates each machining or assembly operation based on both a fixed amount of time and the time required for the robot arm to move an *average* distance necessary for each operation. The distances and times for each machining and assembly operation are given in table 3-3.

Overhanging Board Assembly Sequences

Two alternate assembly sequences for the overhanging board assembly are shown below. AAPF produces the optimal assembly sequence (o1). The second assembly sequence was created by hand and requires a large number of reclamping and tool changing operations. A significant difference of 65% or 76% more time is required for the second assembly sequence depending on the robot arm speed.

Overhanging Board Assembly Sequence o1 Derived by AAPF (Optimal)

Time Cost for Plan			Assembly	
with Robot Arm Speeds			Operation	
of	1.0	and 2.0 ft/sec	Description	
1)	4.5	3.0	Tool Change to	GRIPPER
2)	5.0	5.0	Reclamp	INITIAL-FIXTURE
3)	5.0	3.5	Fetch & Assemble	BLOCK3
4)	5.0	3.5	Fetch & Assemble	BLOCK2
5)	5.0	3.5	Fetch & Assemble	BOLT3
6)	5.0	3.5	Fetch & Assemble	BOLT2
7)	5.0	3.5	Fetch & Assemble	BLOCK1
8)	5.0	3.5	Fetch & Assemble	BLOCK0
9)	5.0	3.5	Fetch & Assemble	BOLT1
10)	5.0	3.5	Fetch & Assemble	BOLT0
11)	5.0	5.0	Reclamp	SUBASSEMBLY
12)	4.5	3.0	Tool Change to	NUT-DRIVER
13)	5.0	3.5	Fetch & Fasten	NUT0
14)	5.0	3.5	Fetch & Fasten	NUT1
15)	5.0	3.5	Fetch & Fasten	NUT2
16)	5.0	3.5	Fetch & Fasten	NUT3
17)	4.5	3.0	Tool Change to	GRIPPER
18)	4.0	2.5	Move Assembly	TO-OUTPUT-BIN
	87.5	63.5	Total assembly times	

Inefficient Overhanging Board Assembly Sequence o2 Derived by Hand

Time Cost for Plan with Robot Arm Speeds of	1.0 and	2.0 ft/sec	Assembly Operation Description	
1)	4.5	3.0	Tool Change to	GRIPPER
2)	5.0	5.0	Reclamp	INITIAL-FIXTURE
3)	5.0	3.5	Fetch & Assemble	BLOCK3
4)	5.0	3.5	Fetch & Assemble	BLOCK2
5)	5.0	3.5	Fetch & Assemble	BOLT3
6)	5.0	5.0	Reclamp	SUBASSEMBLY
7)	4.5	3.0	Tool Change to	NUT-DRIVER
8)	5.0	3.5	Fetch & Fasten	NUT3
9)	4.5	3.0	Tool Change to	GRIPPER
10)	5.0	5.0	Reclamp	SUBASSEMBLY
11)	5.0	3.5	Fetch & Assemble	BOLT2
12)	5.0	5.0	Reclamp	SUBASSEMBLY
13)	4.5	3.0	Tool Change to	NUT-DRIVER
14)	5.0	3.5	Fetch & Fasten	NUT2
15)	4.5	3.0	Tool Change to	GRIPPER
16)	5.0	5.0	Reclamp	SUBASSEMBLY
17)	5.0	3.5	Fetch & Assemble	BLOCK1
18)	5.0	3.5	Fetch & Assemble	BLOCK0
19)	5.0	3.5	Fetch & Assemble	BOLT1
20)	5.0	5.0	Reclamp	SUBASSEMBLY
21)	4.5	3.0	Tool Change to	NUT-DRIVER
22)	5.0	3.5	Fetch & Fasten	NUT1
23)	4.5	3.0	Tool Change to	GRIPPER
24)	5.0	5.0	Reclamp	SUBASSEMBLY
25)	5.0	3.5	Fetch & Assemble	BOLT0
26)	5.0	5.0	Reclamp	SUBASSEMBLY
27)	4.5	3.0	Tool Change to	NUT-DRIVER
28)	5.0	3.5	Fetch & Fasten	NUT0
29)	4.5	3.0	Tool Change to	GRIPPER
30)	4.0	2.5	Move Assembly	TO-OUTPUT-BIN
	144.5	111.5	Total assembly times	

Mechanical Mouse Assembly Sequences

Three assembly sequences for the mechanical mouse are evaluated below. The evaluation of the assembly sequences includes fixturing constraints which AAPF did not consider. This is why the AAPF solution (m3) is very inefficient compared to the other two assembly sequences which were generated by hand. Each assembly sequence is evaluated for two different robot arm speeds of 1.0 and 2.0 feet/sec respectively. The second and third assembly sequences were created by hand and require less assembly time since minimizing the number of reclamping (re-fixturing) steps was a goal of the hand derived sequences.

Efficient Mouse Assembly Sequence m1 Derived by Hand

Time Cost for Plan with Robot Arm Speeds of	1.0 and	2.0 ft/sec	Assembly Operation Description	
1)	4.5	3.0	Tool Change to	GRIPPER
2)	5.0	5.0	Reclamp	PARTIAL-ASSEMBLY
3)	5.0	3.5	Fetch & Assemble	TRACKING-BALL-HOUSING
4)	5.0	3.5	Fetch & Assemble	LEFT-ROLLER-HOLDER
5)	5.0	3.5	Fetch & Assemble	RIGHT-ROLLER-HOLDER
6)	5.0	3.5	Fetch & Assemble	CIRCUIT-BOARD
7)	4.5	3.0	Tool Change to	SCREW-GUN
8)	2.0	2.0	Insert Screw	MAIN-ROLLER-SCREW
9)	2.0	2.0	Insert Screw	RIGHT-ROLLER-SCREW
10)	2.0	2.0	Insert Screw	LEFT-ROLLER-SCREW
11)	4.5	3.0	Tool Change to	GRIPPER
12)	5.0	3.5	Fetch & Assemble	RIGHT-MOUSE-BUTTON
13)	5.0	3.5	Fetch & Assemble	CENTER-MOUSE-BUTTON
14)	5.0	3.5	Fetch & Assemble	LEFT-MOUSE-BUTTON
15)	5.0	3.5	Fetch & Assemble	BLACK-PLASTIC-CASE
16)	5.0	5.0	Reclamp	PARTIAL-ASSEMBLY
17)	5.0	3.5	Fetch & Assemble	MOUSE-CABLE
18)	5.0	3.5	Fetch & Assemble	CIRCUIT-BOARD-SUPPORT
19)	5.0	3.5	Fetch & Assemble	LEFT-ROLLER-BALL
20)	5.0	3.5	Fetch & Assemble	MAIN-TRACKING-BALL
21)	5.0	3.5	Fetch & Assemble	RIGHT-ROLLER-BALL
22)	5.0	3.5	Fetch & Assemble	BASE
23)	4.5	3.0	Tool Change to	SCREW-GUN
24)	2.0	2.0	Insert Screw	LEFT-CASE-SCREW
25)	2.0	2.0	Insert Screw	RIGHT-CASE-SCREW
26)	4.5	3.0	Tool Change to	GRIPPER
27)	4.0	2.5	Move Assembly	TO-FINISHED-BIN
	116.5	86.5	Total assembly times	

Mediocre Mouse Assembly Sequence m2 Derived by Hand

Time Cost for Plan			Assembly	
with Robot Arm Speeds			Operation	
of	1.0	and 2.0 ft/sec	Description	
1)	5.0	5.0	Reclamp	PARTIAL-ASSEMBLY
2)	4.5	3.0	Tool Change to	GRIPPER
3)	5.0	3.5	Fetch & Assemble	CIRCUIT-BOARD
4)	5.0	3.5	Fetch & Assemble	TRACKING-BALL-HOUSING
5)	5.0	5.0	Reclamp	PARTIAL-ASSEMBLY
6)	4.5	3.0	Tool Change to	SCREW-GUN
7)	2.0	2.0	Insert Screw	MAIN-ROLLER-SCREW
8)	4.5	3.0	Tool Change to	GRIPPER
9)	5.0	5.0	Reclamp	PARTIAL-ASSEMBLY
10)	5.0	3.5	Fetch & Assemble	LEFT-ROLLER-HOLDER
11)	5.0	5.0	Reclamp	PARTIAL-ASSEMBLY
12)	4.5	3.0	Tool Change to	SCREW-GUN
13)	2.0	2.0	Insert Screw	LEFT-ROLLER-SCREW
14)	4.5	3.0	Tool Change to	GRIPPER
15)	5.0	5.0	Reclamp	PARTIAL-ASSEMBLY
16)	5.0	3.5	Fetch & Assemble	RIGHT-ROLLER-HOLDER
17)	5.0	5.0	Reclamp	PARTIAL-ASSEMBLY
18)	4.5	3.0	Tool Change to	SCREW-GUN
19)	2.0	2.0	Insert Screw	RIGHT-ROLLER-SCREW
20)	4.5	3.0	Tool Change to	GRIPPER
21)	5.0	3.5	Fetch & Assemble	RIGHT-MOUSE-BUTTON
22)	5.0	3.5	Fetch & Assemble	CENTER-MOUSE-BUTTON
23)	5.0	3.5	Fetch & Assemble	LEFT-MOUSE-BUTTON
24)	5.0	3.5	Fetch & Assemble	BLACK-PLASTIC-CASE
25)	5.0	5.0	Reclamp	PARTIAL-ASSEMBLY
26)	5.0	3.5	Fetch & Assemble	MOUSE-CABLE
27)	5.0	3.5	Fetch & Assemble	CIRCUIT-BOARD-SUPPORT
28)	5.0	3.5	Fetch & Assemble	LEFT-ROLLER-BALL
29)	5.0	3.5	Fetch & Assemble	MAIN-TRACKING-BALL
30)	5.0	3.5	Fetch & Assemble	RIGHT-ROLLER-BALL
31)	5.0	3.5	Fetch & Assemble	BASE
32)	4.5	3.0	Tool Change to	SCREW-GUN
33)	2.0	2.0	Insert Screw	LEFT-CASE-SCREW
34)	2.0	2.0	Insert Screw	RIGHT-CASE-SCREW
35)	4.5	3.0	Tool Change to	GRIPPER
36)	4.0	2.5	Move Assembly	TO-FINISHED-BIN
	159.5	123.5	Total assembly times	

Inefficient Mouse Assembly Sequence m3 Derived by AAPF

	Time Cost for Plan with Robot Arm Speeds of 1.0	and 2.0 ft/sec	Assembly Operation	Description
1)	4.5	3.0	Tool Change to	GRIPPER
2)	5.0	5.0	Reclamp	PARTIAL-ASSEMBLY
3)	5.0	3.5	Fetch & Assemble	CIRCUIT-BOARD
4)	5.0	3.5	Fetch & Assemble	TRACKING-BALL-HOUSING
5)	5.0	5.0	Reclamp	PARTIAL-ASSEMBLY
6)	4.5	3.0	Tool Change to	SCREW-GUN
7)	2.0	2.0	Insert Screw	MAIN-ROLLER-SCREW
8)	4.5	3.0	Tool Change to	GRIPPER
9)	5.0	5.0	Reclamp	PARTIAL-ASSEMBLY
10)	5.0	3.5	Fetch & Assemble	RIGHT-ROLLER-HOLDER
11)	5.0	5.0	Reclamp	PARTIAL-ASSEMBLY
12)	4.5	3.0	Tool Change to	SCREW-GUN
13)	2.0	2.0	Insert Screw	RIGHT-ROLLER-SCREW
14)	4.5	3.0	Tool Change to	GRIPPER
15)	5.0	5.0	Reclamp	PARTIAL-ASSEMBLY
16)	5.0	3.5	Fetch & Assemble	LEFT-ROLLER-HOLDER
17)	5.0	5.0	Reclamp	PARTIAL-ASSEMBLY
18)	4.5	3.0	Tool Change to	SCREW-GUN
19)	2.0	2.0	Insert Screw	LEFT-ROLLER-SCREW
20)	4.5	3.0	Tool Change to	GRIPPER
21)	5.0	5.0	Reclamp	PARTIAL-ASSEMBLY
22)	5.0	3.5	Fetch & Assemble	MOUSE-CABLE
23)	5.0	3.5	Fetch & Assemble	CIRCUIT-BOARD-SUPPORT
24)	5.0	3.5	Fetch & Assemble	MAIN-TRACKING-BALL
25)	5.0	3.5	Fetch & Assemble	RIGHT-ROLLER-BALL
26)	5.0	3.5	Fetch & Assemble	LEFT-ROLLER-BALL
27)	5.0	3.5	Fetch & Assemble	BASE
28)	5.0	5.0	Reclamp	PARTIAL-ASSEMBLY
29)	5.0	3.5	Fetch & Assemble	RIGHT-MOUSE-BUTTON
30)	5.0	3.5	Fetch & Assemble	LEFT-MOUSE-BUTTON
31)	5.0	3.5	Fetch & Assemble	CENTER-MOUSE-BUTTON
32)	5.0	3.5	Fetch & Assemble	BLACK-PLASTIC-CASE
33)	5.0	5.0	Reclamp	PARTIAL-ASSEMBLY
34)	4.5	3.0	Tool Change to	SCREW-GUN
35)	2.0	2.0	Insert Screw	LEFT-CASE-SCREW
36)	2.0	2.0	Insert Screw	RIGHT-CASE-SCREW
37)	4.5	3.0	Tool Change to	GRIPPER
38)	4.0	2.5	Move Assembly	TO-FINISHED-BIN
	169.5	133.5	Total assembly times	

Toy Car Process Plans

Two process plans were derived by hand for the toy car model that was used by the WoodMaster planner in Section 2.1.4 (see Figure 1-1). Both efficient and wasteful sequences are given to show that significant differences in costs can occur. The *bad* plan c2 requires nearly 100% more time to complete than does the efficient plan c1.

Efficient Process Plan c1 for the Car Model Derived by Hand

Time Cost for Plan			Process/Assembly	
with Robot Arm Speeds			Operation	
of	1.0 and	2.0 ft/sec	Description	
1)	5.0	5.0	Reclamp	STOCK-FIXTURE
2)	4.5	3.0	Fetch & Fixture	1/4-INCH-STOCK
3)	4.5	3.0	Tool Change to	CIRCULAR-SAW
4)	2.0	2.0	Make Cut	1/4-INCH-AXLE1
5)	2.0	2.0	Make Cut	1/4-INCH-AXLE2
6)	4.5	3.0	Tool Change to	GRIPPER
7)	4.0	2.5	Move Assembly	REMOVE-STOCK-MATERIAL
8)	4.0	2.5	Move Assembly	AXLE1-TO-BUFFER
9)	4.0	2.5	Move Assembly	AXLE2-TO-BUFFER
10)	4.5	3.0	Tool Change to	GRIPPER
11)	5.0	5.0	Reclamp	STOCK-FIXTURE
12)	4.5	3.0	Fetch & Fixture	2-INCH-STOCK
13)	4.5	3.0	Tool Change to	CIRCULAR-SAW
14)	2.0	2.0	Make Cut	4-1/2INCH-WHEEL
15)	4.5	3.0	Tool Change to	GRIPPER
16)	4.0	2.5	Move Assembly	REMOVE-STOCK-MATERIAL
17)	5.0	5.0	Reclamp	CLAMP-WHEEL
18)	4.5	3.0	Tool Change to	1/4-INCH-DRILL
19)	2.0	2.0	Drill Hole	1/4-INCH-HOLE-IN-WHEELS
20)	4.5	3.0	Tool Change to	GRIPPER
21)	5.0	5.0	Reclamp	4-WHEELS
22)	4.5	3.0	Tool Change to	CIRCULAR-SAW
23)	2.0	2.0	Make Cut	1/2INCH-WHEEL1
24)	2.0	2.0	Make Cut	1/2INCH-WHEEL2
25)	2.0	2.0	Make Cut	1/2INCH-WHEEL3-&-4
26)	4.5	3.0	Tool Change to	GRIPPER
27)	4.0	2.5	Move Assembly	WHEEL2-TO-BUFFER
28)	4.0	2.5	Move Assembly	WHEEL3-TO-BUFFER
29)	4.0	2.5	Move Assembly	WHEEL4-TO-BUFFER
30)	5.0	5.0	Reclamp	WHEEL1
31)	5.0	3.5	Fetch & Assemble	AXLE1
32)	4.0	2.5	Move Assembly	WHEEL1-AXLE1-TO-BUFFER
33)	4.5	3.0	Fetch & Fixture	WHEEL3-FROM-BUFFER
34)	5.0	3.5	Fetch & Assemble	AXLE2
35)	4.0	2.5	Move Assembly	WHEEL3-AXLE2-TO-BUFFER
36)	5.0	5.0	Reclamp	STOCK-FIXTURE
37)	4.5	3.0	Fetch & Fixture	4X1-BOARD-STOCK
38)	4.5	3.0	Tool Change to	CIRCULAR-SAW
39)	2.0	2.0	Make Cut	4X1X10-BODY

40)	2.0	2.0	Make Cut	4X1X4-BODY
41)	4.5	3.0	Tool Change to	GRIPPER
42)	4.0	2.5	Move Assembly	REMOVE-STOCK-MATERIAL
43)	5.0	5.0	Reclamp	CAR-BODY
44)	4.5	3.0	Place	PUT-GLUE-ON-CAR-BODY
45)	4.5	3.0	Tool Change to	GRIPPER
46)	5.0	3.5	Fetch & Assemble	CAR-TOP
47)	5.0	5.0	Reclamp	CAR-BODY-AND-TOP
48)	4.5	3.0	Tool Change to	5/16-INCH-DRILL
49)	2.0	2.0	Drill Hole	5/16-INCH-AXLE-HOLE1
50)	2.0	2.0	Drill Hole	5/16-INCH-AXLE-HOLE2
51)	4.5	3.0	Tool Change to	GRIPPER
52)	5.0	3.5	Fetch & Assemble	AXLE1-WHEEL1-BY-INSERT
53)	5.0	3.5	Fetch & Assemble	AXLE2-WHEEL3-BY-INSERT
54)	5.0	5.0	Reclamp	SUBASSEMBLY
55)	5.0	3.5	Fetch & Assemble	WHEEL2-BY-FORCE-FIT
56)	5.0	3.5	Fetch & Assemble	WHEEL4-BY-FORCE-FIT
57)	4.0	2.5	Move Assembly	MOVE-OUT-FINAL-ASSEMBLY
	231.5	176.0	Total assembly times	

Bad Process Plan c2 for the Car Model Derived by Hand

Time Cost for Plan with Robot Arm Speeds of	1.0 and	2.0 ft/sec	Process/Assembly Operation Description	
1)	4.5	3.0	Tool Change to	GRIPPER
2)	5.0	5.0	Reclamp	STOCK-FIXTURE
3)	4.5	3.0	Fetch & Fixture	2-INCH-STOCK
4)	4.5	3.0	Tool Change to	CIRCULAR-SAW
5)	2.0	2.0	Make Cut	1/2-INCH-WHEEL1
6)	4.5	3.0	Tool Change to	GRIPPER
7)	4.0	2.5	Move Assembly	REMOVE-STOCK-MATERIAL
8)	4.0	2.5	Move Assembly	WHEEL1-TO-BUFFER
9)	5.0	5.0	Reclamp	STOCK-FIXTURE
10)	4.5	3.0	Fetch & Fixture	4X1-BOARD-STOCK
11)	4.5	3.0	Tool Change to	CIRCULAR-SAW
12)	2.0	2.0	Make Cut	4X1X10-BODY
13)	4.5	3.0	Tool Change to	GRIPPER
14)	4.0	2.5	Move Assembly	REMOVE-STOCK-MATERIAL
15)	4.0	2.5	Move Assembly	CAR-BODY-TO-BUFFER
16)	5.0	5.0	Reclamp	GET-WHEEL-CLAMP
17)	4.5	3.0	Fetch & Fixture	UNFINISHED-WHEEL1
18)	4.5	3.0	Tool Change to	1/4-INCH-DRILL
19)	2.0	2.0	Drill Hole	1/4-INCH-HOLE-IN-WHEEL1
20)	4.5	3.0	Tool Change to	GRIPPER
21)	4.0	2.5	Move Assembly	WHEEL1-TO-BUFFER
22)	5.0	5.0	Reclamp	GET-CAR-BODY-CLAMP
23)	4.5	3.0	Fetch & Fixture	CAR-BODY
24)	4.5	3.0	Tool Change to	5/16-INCH-DRILL
25)	2.0	2.0	Drill Hole	5/16-INCH-AXLE-HOLE

26)	4.5	3.0	Tool Change to	GRIPPER
27)	4.0	2.5	Move Assembly	CAR-BODY-TO-BUFFER
28)	5.0	5.0	Reclamp	STOCK-FIXTURE
29)	4.5	3.0	Fetch & Fixture	2-INCH-STOCK
30)	4.5	3.0	Tool Change to	CIRCULAR-SAW
31)	2.0	2.0	Make Cut	1/2INCH-WHEEL2
32)	4.5	3.0	Tool Change to	GRIPPER
33)	4.0	2.5	Move Assembly	REMOVE-STOCK-MATERIAL
34)	4.0	2.5	Move Assembly	WHEEL2-TO-BUFFER
35)	5.0	5.0	Reclamp	STOCK-FIXTURE
36)	4.5	3.0	Fetch & Fixture	4X1-BOARD-STOCK
37)	4.5	3.0	Tool Change to	CIRCULAR-SAW
38)	2.0	2.0	Make Cut	4X1X4-BODY
39)	4.5	3.0	Tool Change to	GRIPPER
40)	4.0	2.5	Move Assembly	REMOVE-STOCK-MATERIAL
41)	5.0	5.0	Reclamp	CAR-TOP
42)	4.5	3.0	Tool Change to	GRIPPER
43)	4.0	2.5	Move Assembly	CAR-TOP-TO-BUFFER
44)	5.0	5.0	Reclamp	GET-WHEEL-CLAMP
45)	4.5	3.0	Fetch & Fixture	UNFINISHED-WHEEL2
46)	4.5	3.0	Tool Change to	1/4-INCH-DRILL
47)	2.0	2.0	Drill Hole	1/4-INCH-HOLE-IN-WHEEL2
48)	4.5	3.0	Tool Change to	GRIPPER
49)	4.0	2.5	Move Assembly	WHEEL2-TO-BUFFER
50)	5.0	5.0	Reclamp	CAR-BODY
51)	4.5	3.0	Tool Change to	5/16-INCH-DRILL
52)	2.0	2.0	Drill Hole	5/16-INCH-AXLE-HOLE
53)	4.5	3.0	Tool Change to	GRIPPER
54)	4.0	2.5	Move Assembly	CAR-BODY-TO-BUFFER
55)	5.0	5.0	Reclamp	STOCK-FIXTURE
56)	4.5	3.0	Fetch & Fixture	2-INCH-STOCK
57)	4.5	3.0	Tool Change to	CIRCULAR-SAW
58)	2.0	2.0	Make Cut	1/2INCH-WHEEL
59)	4.5	3.0	Tool Change to	GRIPPER
60)	4.0	2.5	Move Assembly	REMOVE-STOCK-MATERIAL
61)	4.0	2.5	Move Assembly	WHEEL3-TO-BUFFER
62)	5.0	5.0	Reclamp	STOCK-FIXTURE
63)	4.5	3.0	Fetch & Fixture	1/4-INCH-STOCK
64)	4.5	3.0	Tool Change to	CIRCULAR-SAW
65)	2.0	2.0	Make Cut	1/4-INCH-AXLE
66)	4.5	3.0	Tool Change to	GRIPPER
67)	4.0	2.5	Move Assembly	AXLE1-TO-BUFFER
68)	5.0	5.0	Reclamp	GET-WHEEL-CLAMP
69)	4.5	3.0	Fetch & Fixture	UNFINISHED-WHEEL3
70)	4.5	3.0	Tool Change to	1/4-INCH-DRILL
71)	2.0	2.0	Drill Hole	1/4-INCH-HOLE-IN-WHEEL3
72)	4.5	3.0	Tool Change to	GRIPPER
73)	4.0	2.5	Move Assembly	WHEEL3-TO-BUFFER
74)	5.0	5.0	Reclamp	STOCK-FIXTURE
75)	4.5	3.0	Fetch & Fixture	2-INCH-STOCK
76)	4.5	3.0	Tool Change to	CIRCULAR-SAW
77)	2.0	2.0	Make Cut	1/2INCH-WHEEL3

78)	4.5	3.0	Tool Change to	GRIPPER
79)	4.0	2.5	Move Assembly	REMOVE-STOCK-MATERIAL
80)	4.0	2.5	Move Assembly	WHEEL3-TO-BUFFER
81)	5.0	5.0	Reclamp	STOCK-FIXTURE
82)	4.5	3.0	Fetch & Fixture	1/4-INCH-STOCK
83)	4.5	3.0	Tool Change to	CIRCULAR-SAW
84)	2.0	2.0	Make Cut	1/4-INCH-AXLE2
85)	4.5	3.0	Tool Change to	GRIPPER
86)	4.0	2.5	Move Assembly	AXLE2-TO-BUFFER
87)	5.0	5.0	Reclamp	GET-WHEEL-CLAMP
88)	4.5	3.0	Fetch & Fixture	UNFINISHED-WHEEL
89)	4.5	3.0	Tool Change to	1/4-INCH-DRILL
90)	2.0	2.0	Drill Hole	1/4-INCH-HOLE-IN-WHEEL
91)	4.5	3.0	Tool Change to	GRIPPER
92)	4.0	2.5	Move Assembly	WHEEL-TO-BUFFER
93)	5.0	5.0	Reclamp	WHEEL-FIXTURE
94)	4.5	3.0	Fetch & Fixture	WHEEL1
95)	5.0	3.5	Fetch & Assemble	AXLE1-BY-FORCE-FIT
96)	4.0	2.5	Move Assembly	AXLE-WHEEL-TO-BUFFER
97)	5.0	5.0	Reclamp	CAR-BODY-FIXTURE
98)	4.5	3.0	Fetch & Fixture	CAR-BODY
99)	5.0	3.5	Fetch & Assemble	CAR-TOP
100)	4.0	2.5	Move Assembly	BODY-TOP-TO-BUFFER
101)	5.0	5.0	Reclamp	WHEEL-FIXTURE
102)	4.5	3.0	Fetch & Fixture	WHEEL3
103)	5.0	3.5	Fetch & Assemble	AXLE2-BY-FORCE-FIT
104)	4.0	2.5	Move Assembly	AXLE-WHEEL-TO-BUFFER
105)	5.0	5.0	Reclamp	CAR-BODY-FIXTURE
106)	4.5	3.0	Fetch & Fixture	CAR-BODY
107)	5.0	3.5	Fetch & Assemble	AXLE1-WHEEL1
108)	5.0	3.5	Fetch & Assemble	WHEEL2-BY-FORCE-FIT
109)	5.0	3.5	Fetch & Assemble	AXLE2-WHEEL3
110)	5.0	3.5	Fetch & Assemble	WHEEL4-BY-FORCE-FIT
111)	4.0	2.5	Move Assembly	MOVE-OUT-FINAL-ASSEMBLY
	465.5	348.5	Total assembly times	

Block with 6 Holes - Two Process Plans

Several machining and assembly sequences are given for the creation of a block with 6 holes in it (similar to the block shown in Figure 2-10). Both of the process plans were derived by hand to show that the *inefficient* plan b2 requires over 100% more time to complete than does the efficient plan b1.

Efficient Process Plan b1 for the Block Derived by Hand

Time Cost for Plan with Robot Arm Speeds of	1.0 and	2.0 ft/sec	Process/Assembly Operation Description	
1)	4.5	3.0	Tool Change to	GRIPPER
2)	5.0	5.0	Reclamp	INITIAL-FIXTURE
3)	5.0	3.5	Fetch & Assemble	BLOCK
4)	4.5	3.0	Tool Change to	1/2DRILL
5)	2.0	2.0	Drill Hole	HOLE-1
6)	2.0	2.0	Drill Hole	HOLE-2
7)	2.0	2.0	Drill Hole	HOLE-3
8)	4.5	3.0	Tool Change to	GRIPPER
9)	5.0	5.0	Reclamp	BLOCK
10)	4.5	3.0	Tool Change to	1/2DRILL
11)	2.0	2.0	Drill Hole	HOLE-4
12)	2.0	2.0	Drill Hole	HOLE-5
13)	2.0	2.0	Drill Hole	HOLE-6
14)	4.5	3.0	Tool Change to	GRIPPER
15)	4.0	2.5	Move Assembly	REMOVE-FINISHED-BLOCK
	53.5	43.0	Total assembly times	

Bad Process Plan b2 for the Block Derived by Hand

Time Cost for Plan			Process/Assembly	
with Robot Arm Speeds			Operation	
of	1.0 and	2.0 ft/sec	Description	
1)	4.5	3.0	Tool Change to	GRIPPER
2)	5.0	5.0	Reclamp	INITIAL-FIXTURE
3)	5.0	3.5	Fetch & Assemble	BLOCK
4)	4.5	3.0	Tool Change to	1/2DRILL
5)	2.0	2.0	Drill Hole	HOLE-1
6)	4.5	3.0	Tool Change to	GRIPPER
7)	5.0	5.0	Reclamp	BLOCK
8)	4.5	3.0	Tool Change to	1/2DRILL
9)	2.0	2.0	Drill Hole	HOLE-4
10)	4.5	3.0	Tool Change to	GRIPPER
11)	5.0	5.0	Reclamp	BLOCK
12)	4.5	3.0	Tool Change to	1/2DRILL
13)	2.0	2.0	Drill Hole	HOLE-2
14)	4.5	3.0	Tool Change to	GRIPPER
15)	5.0	5.0	Reclamp	BLOCK
16)	4.5	3.0	Tool Change to	1/2DRILL
17)	2.0	2.0	Drill Hole	HOLE-5
18)	4.5	3.0	Tool Change to	GRIPPER
19)	5.0	5.0	Reclamp	BLOCK
20)	4.5	3.0	Tool Change to	1/2DRILL
21)	2.0	2.0	Drill Hole	HOLE-3
22)	4.5	3.0	Tool Change to	GRIPPER
23)	5.0	5.0	Reclamp	BLOCK
24)	4.5	3.0	Tool Change to	1/2DRILL
25)	2.0	2.0	Drill Hole	HOLE-6
26)	4.5	3.0	Tool Change to	GRIPPER
27)	4.0	2.5	Move Assembly	REMOVE-FINISHED-BLOCK
	109.5	87.0	Total assembly times	

Appendix III: Comparison of the Oracle† to Enumeration

Table A-1 shows run time data for examples processed by the first version of the Oracle algorithm as described in [Mil89b, Mil90a]. The Oracle algorithm was coded in Common Lisp and the run times are for a Sun 4/330 running Unix. The enumeration algorithm is a modified topo-sort algorithm that uses backtracking to find every valid ordering of the tasks. Any enumeration algorithm will require at least time proportional to the number of orderings possible. The run times for enumeration are roughly proportional to the total number of orders and are dramatically higher than the Oracle run times.

Table A-1: The Oracle vs. Enumeration for Selected Example Graphs

Data set	No. tasks, constraints	No. of orders‡	Oracle† run time	Enumeration run time
Morning ex	8 , 5	1120	4.9 msec	495 msec
Desk Lamp	8 , 6	336	5.3 msec	180 msec
Toy Car	12 , 11	99792	7.4 msec	62100 msec
Door Lock	14 , 15	308880	10.7 msec	231000 msec
Strainer	31 , 30	7.5×10^{22}	63.9 msec	*intractable*

‡ The *Oracle Algorithm* calculated the exact number of orders possible for every example in this table.

The Oracle was also tested on several structured synthetic data sets. Some of these results are shown in Tables A-2 and A-3 [Mil89b]. Table A-2 shows the results for tasks without any precedence constraints. These trivial cases have a total of N factorial different orderings as the Oracle correctly determined. Any enumeration algorithm would have an exponential time complexity for this type of problem so no enumeration algorithm run times are given. The results in Table A-1 and Table A-2 combined show that for even a modest number of loosely constrained tasks performing the enumeration of all

† Although the tables presented in this appendix were derived using the first version of the Oracle algorithm (described in [Mil89b, Mil90a]), the behavior of both versions of the Oracle algorithm is similar, so all the conclusions in this appendix hold for the current version of the Oracle algorithm as well.

sequences is impractical.

Table A-2: The Oracle vs. Enumeration of Unconstrained Tasks

No. of vertices	No. of tasks	No. of constraints	No. of orders‡	Oracle† run time
5	5	0	120	4.5 msec
10	10	0	3.6 million	9.4 msec
15	15	0	1.3×10^{12}	18.7 msec
20	20	0	2.4×10^{18}	37.8 msec
40	40	0	8.2×10^{47}	77.3 msec

Table A-3 shows the results of the Oracle algorithm on seven different graphs containing nested simple folds. The first graph in the table consists of two vertices in parallel, each with the same parent and child (a single *simple fold*). The graphs 2 through 9 were created by randomly replacing edges in the previous graph by two vertices in parallel to create a nested simple fold. Table A-3 demonstrates that the first version of the Oracle algorithm works in low order polynomial time for simple fold graphs.

Table A-3: The Oracle vs. Enumeration of Nested Simple Fold Graphs

Data set	N = No. of tasks	M = No. of constraints	Total number of orders‡	Oracle† run time
1st	4	4	2	3.2 msec
2nd	6	7	8	4.7 msec
3rd	8	10	48	7.2 msec
4th	14	19	96768	14.8 msec
5th	28	40	1.0×10^{15}	38.5 msec
9th	168	250	2.9×10^{161}	520.0 msec

† The first version of the Oracle algorithm was used for these tables [Mil90a]. Enumeration run times are not given since they would be proportional to the number of different task orderings.

‡ The *Oracle Algorithm* calculated the exact number of orders possible for each table entry.

Bibliography

Bibliography

[Aho72] A. V. Aho, M. R. Garey, and J. D. Ullman, *The Transitive Reduction of a Directed Graph*, Siam J. Computing, Vol. 1, No. 2, June 1972, pages 131-137.

[Aho74] Alfred Aho, John Hopcroft, Jeffrey Ulman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Massachusetts, 1974, 470 pages.

[Akag80] Fumio Akagi, Hirokazu Osaki, Susumu Kikuchi, *The Method of Analysis of Assembly Work Based on the Fastener Method*, Bulletin of the JSME, Vol. 23, No. 184, October 1980, pages 1670-1675.

[Alte86] Richard Alterman, *An Adaptive Planner*, 5th National Conference on Artificial Intelligence (AAAI-86), Philadelphia, Pennsylvania, August 11-15 1986, pages 65-69.

[Arda87] David Ardayfio, Donling Jing, Matthew Hays, *Prototype Expert Systems for Engineering Design and Manufacturing Automation*, Proc. SAE/ESD Computer Graphics 1987, SAE/ESD, April 7-9 1987 Cobo Hall Detroit, pages 207-222.

[Atki86] M. D. Atkinson, H. W. Chang, *Extensions of Partial Orders of Bounded Width*, Congressus Numerantium, Vol. 52, 1986, pages 21-35.

[Bere86] Hamid Berenji, Behrokh Khoshnevis, *Use of Artificial Intelligence in Automated Process Planning*, Computers in Mechanical Engineering, September 1986, pages 47-55.

[Berr84] Gayle L. Berry, *Computer-Aided Production Engineering the Integration of CAPP, Engineering and Manufacturing*, Proceedings of Autofact #6, Anaheim, California, October 1-4 1984, pages 14-1 to 14-9.

[Boer88] J. Boerma, H. Kals, *FIXES, a System for Automatic Selection of Set-Ups and Design of Fixtures*, Annals of the CIRP, Vol. 37, No. 1, 1988, pages 443-446.

[Boot82] Geoffrey Boothroyd, Corrado Poli, and Laurence E. Murch, *Automatic Assembly*, Marcel Kekker, Inc., New York, 1982, 378 pages.

[Bour84] A. Bourjault, *Contribution a une Approche Methodologique de l'Assembly Automatise: Elaboration Automatigique des Sequences Operatoires,* PhD Thesis, University of Franche-Comte, November 1984.

[Brdy87] Herb Brody, *CAD Meets CAM,* High Technology, May 1987, pages 12-18.

[Brod87] Steven Brodd, *A Strategy Planner for NASA Robotics Applications,* (Workshop) on Spatial Reasoning and Multi-Sensor Fusion, Goddard Space Flight Center, Lanham MD, October 1987, 13 pages.

[Brow82] Christopher Brown, *PADL-2: A Technical Summary,* IEEE Computer Graphics and Applications, Vol. 2, No. 2, March 1982, pages 69-84.

[Cama87] L. Camarinha-Matos, *Plan Generation in Robotics,* Robotics, Vol. 3, No. 3 & 4, September-December 1987, pages 291-328.

[Chan85] Tien-Chien Chang, Richard A. Wysk, *An Introduction to Automated Process Planning Systems,* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1985, 230 pages.

[Chan89] Kuo-Chu Chang, Robert Fung, *Node Aggregation for Distributed Inference in Bayesian Networks,* IJCAI-89 Eleventh International Joint Conference on Artificial Intelligence, Morgan Kaufmann Publishers, Inc., Detroit Michigan, August 20-25 1989, Vol. 1, pages 265-270.

[Crow84] James Crowley, *A Computational Paradigm for Three Dimensional Scene Analysis,* Technical Report from the Robotics Institute, CMU, No. CMU-RI-TR-84-11, April 1984, 18 pages.

[DeFa87] T. De Fazio, D. Whitney, *Simplified Generation of All Mechanical Assembly Sequences,* IEEE Journal of Robotics and Automation, December 1987, Vol. 3, pages 640-658.

[Desc81] Yannick Descotte, Jean-Claude Latombe, *GARI: A Problem Solver that Plans How to Machine Mechanical Parts,* Proceedings of the Seventh International Joint Conference on Artificial Intelligence, August 24-28 1981, Vol. 2, pages 766-772.

[Dixo86] John Dixon, *Artificial Intelligence and Design: A Mechanical Engineering View,* 5th National Conference on Artificial Intelligence (AAAI-86), Philadelphia, Pennsylvania, August 11-15 1986, pages 872-877.

[Dona87] Bruce R. Donald, *A Search Algorithm for Motion Planning with Six Degrees of Freedom,* Artificial Intelligence, Vol. 31, 1987, pages 295-353.

[Engl86] Paul Englert, Paul K. Wright, *Applications of Artificial Intelligence and the Design of Fixtures for Automated Manufacturing,* IEEE Proceedings of the International Conference on Robotics and Automation, April 7-10 1986, Vol. 1, pages 345-351.

[Fers86] F. Ferstenberg, K. K. Wang, J. Muckstadt, *Automatic Generation of Optimized 3-Axis NC Programs Using Boundary Files,* IEEE Proceedings of the International Conference on Robotics and Automation, April 7-10 1986, Vol. 1, pages 325-332.

[Fox85] B. R. Fox, K. G. Kempf, *Opportunistic Scheduling for Robotic Assembly,* IEEE International Conference on Robotics and Automation, IEEE Computer Society, 1985, pages 880-889.

[Gare79] Michael R. Garey, David S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, New York, 1979, 340 pages.

[Groo80] Mikell Groover, *Automation, Production Systems, and Computer-Aided Manufacturing,* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1980, 601 pages.

[Groo83] Mikell P. Groover, *Fundamental Operations*, IEEE Spectrum, Vol. 20, No. 5, May 1983, pages 65-69.

[Habi87] M. Habib, R. H. Mohring, *On some Complexity Properties of N-Free Posets and Posets with Bounded Decomposition Diameter*, Discrete Mathematics, Vol. 63, 1987, pages 157-182.

[Hara69] Frank Harary, *Graph Theory*, Addison-Wesley, Reading, Mass., 1969, 274 pages.

[Hera87] Sunderesh Heragu, Andrew Kusiak, *Analysis of Expert Systems in Manufacturing Design,* IEEE Transactions on Systems, Man and Cybernetics, Vol. SMC-17, No. 6, November/December 1987, pages 898-912.

[Hof89a] Richard L. Hoffman, *Automated Assembly in a CSG Domain,* IEEE International Conference on Robotics and Automation Conference, Scottsdale Arizona, May 14-18 1989, Vol. 1, pages 210-215.

[Hof89b] Richard L. Hoffman, *Assembly Planning for CSG Objects,* Technical Report #NRTC-8916, Northrop Research and Technology Center, May 1989.

[Hof90a] Richard L. Hoffman, *Assembly Planning for B-Rep Objects*, Proceedings of the 2nd International Conference on Computer Integrated Manufacturing, Rensselaer Polytechnic Institute, Troy NY, May 21-23 1990, pages 314-321.

[Hof90b] Richard L. Hoffman, *Automated Assembly Planning for B-rep Products*, To appear: IEEE International Conference on Systems Engineering, Pittsburgh PA, August 9-11 1990.

[Home86] Luiz Homem-de-Mello, Arthur C. Sanderson, *AND / OR Graph Representation of Assembly Plans,* CMU Report, No. CMU-RI-TR-86-8, Carnegie-Mellon University, Pittsburgh, Pennsylvania, April 1986, 18 pages.

[Home89] L. S. Homem de Mello, *Task Sequence Planning for Robotic Assembly* Carnegie Mellon University Ph.D. Thesis, May 1989, 212 pages.

[Home90] L.S. Homem de Mello, A. C. Sanderson, *Evaluation and Selection of Assembly Plans* IEEE International Conference on Robotics and Automation, Cincinnati, Ohio, May 13-18 1990, Vol. 3, pages 1588-1593.

[Horo76] Ellis Horowitz, Sartaj Sahni, *Fundamentals of Data Structures,* Computer Science Press, Inc., Woodland Hills, California, 1976, 564 pages.

[Horo89] Ellis Horowitz, Sartaj Sahni, *Fundamentals of Computer Algorithms,* Computer Science Press, Rockville, Maryland, 1989, 626 pages.

[Huan89] Y. F. Huang, C.S.G. Lee, *Precedence Knowledge in Feature Mating Operation Assembly Planning,* IEEE International Conference on Robotics and Automation, Scottsdale, Arizona, May 14-18 1989, Vol. 1, pages 216-221.

[Huan90] Y. F. Huang, C. S. G. Lee, *An Automatic Assembly Planning System* IEEE International Conference on Robotics and Automation Cincinnati Ohio, May 13-18 1990, pages 1594-1599.

[Hutc90] Seth A. Hutchinson, Avinash C. Kak, *Spar: A Planner that Satisfies Operational and Geometric Goals in Uncertain Environments*, AI Magazine, Vol. 11, No. 1, Spring 1990, pages 30-61.

[Kak86] A. Kak, K. Boyer, C. Chen, R. Safranek, and H. Yang, *A Knowledge-Based Robotic Assembly Cell,* IEEE Expert, Spring, 1986, pages 63-83.

[Kemp87] Alfons Kemper, Mechtild Wallrath, *An Analysis of Geometric Modeling in Database Systems,* ACM Computing Surveys, Vol. 19, No. 1, March 1987, pages 47-91.

[Ko87] Heedong Ko, Kunwoo Lee, *Automatic Assembling Procedure Generation from Mating Conditions,* Computer-Aided Design, Vol. 19, No. 1, January-February 1987, pages 3-10.

[Kuth83] Mike Kutcher, Eli Gorin, *Moving Data, not Paper, Enhances Productivity,* IEEE Spectrum, Vol. 20, No. 5, May 1983, pages 40-43.

[Laug86] C. Laugier, P. Theveneau, *Planning Sensor-Based Motions for Part-Mating Using Geometric Reasoning Techniques,* Proc. 7th European Conf. on Artificial Intelligence, July 1986, pages 494-506.

[Lieb77] L. Lieberman, M. A. Wesley, *AUTOPASS: An Automatic Programming System for Computer Controlled Mechanical Assembly,* IBM Journal Research and Development, July 1977, pages 321-333.

[Lind83] Roy Lindberg, *Processes and Materials of Manufacture,* Allyn and Bacon, Inc., Boston, 1988, 849 pages.

[Loza76] T. Lozano-Perez, *The Design of a Mechanical Assembly System,* M.I.T. Artificial Intell. Lab., Rep. TR-397, December 1976.

[Maki88] Hiroshi Makino, Nobuyuki Fujita, *Assembly of Blocks by Autonomous Assembly Robot with Intelligence (ARI),* Annals of the CIRP, Vol. 37, No. 1, Tokyo, Japan, August 22-27 1988, pages 33-36.

[Mats82] K. Matsushima, N. Okada, T. Sata, *The Integration of CAD and CAM by Application of Artificial Intelligence Techniques*, CIRP Annals, Vol. 31/1, 1982, pages 329-332.

[McLe83] Charles McLean, Mary Mitchell, Edward Barkmeyer, *A Computer Architecture for Small-Batch Manufacturing* IEEE Spectrum, Vol. 20, No. 5, May 1983, pages 59-64.

[Mill88] Joseph M. Miller, *WoodMaster: A Process Planner for Wooden Objects Made up of Block and Cylindrical Components,* Case Center Technical Report, April 1988, 65 pages.

[Mil89a] Joseph M. Miller, Richard L. Hoffman, *Automatic Assembly Planning with Fasteners*, IEEE International Conference on Robotics and Automation Conference, Scottsdale Arizona, May 14-18 1989, Vol. 1, pages 69-74.

[Mil89b] *On the Number of Linear Extensions in a Precedence Graph,* Joseph M. Miller, George C. Stockman, Michigan State University, Department of Computer Science, Technical Report #PRIP-89-7, 62 pages.

[Mil90a] Joseph Miller, George Stockman, *On the Number of Linear Extensions in a Precedence Graph,* IEEE International Conference on Robotics and Automation, Cincinnati Ohio, May 13-18 1990, pages 2136-2141.

[Mil90b] Joseph M. Miller, George C. Stockman, *Precedence Constraints and Tasks: How Many Task Orderings,* To appear in IEEE International Conference on Systems Engineering, Pittsburgh Pennsylvania, August 9-11 1990, pages 408-411.

[Myru83] M. Myrup Andreasen, S. Kahler, T. Lund, *Design for Assembly,* IFS (Publications) Ltd. and Springer-Verlag, Berlin, New York, 1983, 189 pages.

[NaCh85] Dana Nau, Tien-Chien Chang, *Hierarchical Representation of Problem-Solving Knowledge in a Frame-Based Process Planning System,* Production Engineering Conference at ASME Winter Annual Meeting, November 1985, Miami Beach, pages 65-71.

[Nau86] Dana Nau, Michael Gray, *Hierarchical Knowledge Clustering: A New Representation for Problem Solving Knowledge,* 5th National Conference on Artificial Intelligence (AAAI-86), March 1986, 8 pages.

[Nevi89] James L. Nevins, Daniel E. Whitney, editors, *Concurrent Design of Products & Processes,* McGraw-Hill, New York, 1989, 538 pages.

[Nils80] Nils Nilson, *Principles of AI,* Book chapters 7 and 8, Tioga Publishing Co. Palo Alto CA, Palo Alto CA, 1980, pages 275-357.

[Palm87] Richard Stuart Palmer, *Computational Complexity of Motion and Stability of Polygons,* Cornell University Ph.D. Thesis, UMI Dissertation Service, 1987, 128 pages.

[Phil85] R. Phillips, C. Mouleeswaran, *A Knowledge-Based Approach to Generative Process Planning,* AUTOFACT, November 6 1985, pages 10-1 - 10-15.

[Popp90] Robin J. Popplestone, Yanzi Liu, Rich Weiss, *A Group Theoretic Approach to Assembly Planning,* AI Magazine, Vol. 11, No. 1, Spring 1990, pages 82-97.

[Requ80] Aristides Requicha, *Representations for Rigid Solids: Theory, Methods, and Systems,* Computing Surveys, Vol. 12, No. 4, December 1980, pages 437-464.

[Requ82] A. Requicha, H. B. Voelcker, *Solid Modeling: A Historical Summary and Contemporary Assessment,* IEEE Computer Graphics and Applications, IEEE, March 1982, pages 9-24.

[Reym87] G. Reyman, *Design of Assembly Systems,* The International Journal of Advanced Manufacturing Technology, Vol. 2, No. 3, August 1987, pages 13-21.

[Rich83] Elaine Rich, *Artificial Intelligence,* McGraw-Hill, New York, 1983, 436 pages.

[Rich86] John Richardson, *Implementing an Expert System for Process Planning*, Society of Manufacturing Engineers Conference, March 1986, Paper No. 860339, pages 43-50.

[Riva84] Ivan Rival, *Linear Extensions of Finite Ordered Sets,* Annals of Discrete Mathematics, Vol 23 1984, pages 355-370.

[Rych88] Michael D. Rychener, editor, *Expert Systems for Engineering Design*, Academic Press, Inc., 1988, 306 pages.

[Sand90] Arthur C. Sanderson, Luiz S. Homem de Mello, Hui Zhang, *Assembly Sequence Planning*, AI Magazine, Vol. 11, No. 1, Spring 1990, pages 62-81.

[Scha77] Roger Shank, Robert Abelson, *Scripts, Plans, Goals and Understanding,* Lawrence Erlbaum, Hillsdale, N.J., 1977.

[Seda87] S. Sedas, S. Talukdar, *A Disassembly Planner for Redesign,* Tech. Rep. EDRC-05-16-87, Engineering Design Research Center, Carnegie Mellon University, May 1987, 6 pages.

[Seki88] Yukiko Sekine, John J. Mills, *Generation of Assembly Sequences in Assembly Process Planning,* AAAI-88 Workshop on Manufacturing Planning and Scheduling, St. Paul, August 1988, 4 pages.

[Shar86] Micha Sharir, Amir Schorr, *On Shortest Paths in Polyhedral Spaces*, Siam J. Computing, Vol. 15, No. 1, February 1986, pages 193-215.

[Stan86] Richard P. Stanley, *Enumerative Combinatorics,* Wadsworth & Brooks/Cole, Monterey, CA, Vol. 1, 1986, 306 pages.

[Ste81a] Mark Stefik, *Planning with Constraints, (MOLGEN: Part 1)*, Artificial Intelligence, North-Holland Publishing Company, Vol. 16, No. 2, 1981, pages 111-139.

[Ste81b] Mark Stefik, *Planning and Meta-Planning (MOLGEN: Part 2)*, Artificial Intelligence (An International Journal), North-Holland Publishing Company, Vol. 16, No. 2, 1981, pages 141-169.

[Take83] H. Takeyama, H. Sekiguchi, T. Kojima, K. Inoue, T. Honda, *Study on Automatic Determination of Assembly Sequence*, Annals of the CIRP, Harrogate, England, Vol. 32, No. 1, August 22-27 1983, pages 371-374.

[Tilo84] Robert Tilove, Vadim Shapiro, Mary S. Pickett, *Modeling and Analysis of Robot Work Cells in Roboteach*, GM Research Publication, No. GMR-4661, GM, March 27 1984, 33 pages.

[Tsat87] Costas Tsatsoulis, R. Kashyap, *Using Dynamic Memory Structures in Planning and its Application to Manufacturing*, Technical Report, Purdue Engineering Research Center for IMS, No. TR-ERC 87-9, West Lafayette, Indiana 47907, July 1987, 157 pages.

[Well72] Mark B. Wells, *Elements of Combinatorial Computing*, Pergamon Press, New York, 1971, 258 pages.

[Whit88] Daniel Whitney, Thomas De Fazio, Richard Gustavson, Stephen Graves, Kurt Cooprider, Charles Klein, Mancheung Lui, Suguna Pappu, *Computer Aided Design of Flexible Assembly Systems: Final Report*, Charles Stark Draper Laboratory Technical Report No. CDSL-R-2033, January 1988, 100 pages.

[Wins84] Patrick Winston, *Artificial Intelligence*, Addison-Wesley, Reading Massachusetts, 1984, 524 pages.

[Wink90] Peter Winkler, Graham Brightwell, *Counting Linear Extensions Is #P-Complete*, DIMACS Technical Report 90-49, Rutgers University, July 1990, 18 pages.

[Wolt89] Jan Wolter, *On the Automatic Generation of Assembly Plans*, IEEE International Conference on Robotics and Automation Conference, Scottsdale Arizona, May 14-18 1989, Vol. 1, pages 62-68.

[Wolt90] Jan Wolter, *A Survey of Enumerative Data Structures for Assembly Planning,* Technical Report 90-006, Computer Science Department, Texas A&M University, March 1990, 32 pages.

[Woo87] Tony Woo, *Automatic Disassembly,* Proc. SAE/ESD Computer Graphics 1987, SAE/ESD, Cobo Hall Detroit, April 7-9 1987, pages 163-166.

[Wu86] Ching-Farn Wu, Anthony S. Wojcik, Lionel M. Ni, *CMOS Circuit Representation, Verification and Synthesis Using Automated Reasoning,* MSU Technical Report, No. MSU-ENGR-86-022, MSU, November 19 1986. 57 pages.

[Yama87] Seiji Yamada, Norihiro Abe, Saburo Tsuji, *Construction of a Consulting System from Structural Description of a Mechanical Object,* IEEE International Conference on Robotics and Automation, March 1987, pages 1413-1418.